Tamper Evident Microprocessors

Adam Waksman
Department of Computer Science
Columbia University
New York, USA
waksman@cs.columbia.edu

Simha Sethumadhavan
Department of Computer Science
Columbia University
New York, USA
simha@cs.columbia.edu

***Abstract*—Most security mechanisms proposed to date unquestioningly place trust in microprocessor hardware. This trust, however, is misplaced and dangerous because microprocessors are vulnerable to insider attacks that can catastrophically compromise security, integrity and privacy of computer systems. In this paper, we describe several methods to strengthen the fundamental assumption about trust in microprocessors. By employing practical, lightweight attack detectors within a microprocessor, we show that it is possible to protect against malicious logic embedded in microprocessor hardware.**

We propose and evaluate two area-efficient hardware methods — TRUSTNET and DATAWATCH — that detect attacks on microprocessor hardware by knowledgeable, malicious insiders. Our mechanisms leverage the fact that multiple components within a microprocessor (*e.g.,* fetch, decode pipeline stage etc.) must necessarily coordinate and communicate to execute even simple instructions, and that any attack on a microprocessor must cause erroneous communications between microarchitectural subcomponents used to build a processor. A key aspect of our solution is that TRUSTNET and DATAWATCH are themselves highly resilient to corruption. We demonstrate that under realistic assumptions, our solutions can protect pipelines and on-chip cache hierarchies at negligible area cost and with no performance impact. Combining TRUSTNET and DATAWATCH with prior work on fault detection has the potential to provide complete coverage against a large class of microprocessor attacks.[1]

***Index Terms*—hardware security, backdoors, microprocessors, security based on causal structure and division of work.**

I. INTRODUCTION

One of the key challenges in trustworthy computing is establishing trust in the microprocessors that underlie all modern IT. The root of trust in all software systems rests on microprocessors because all software is executed by a microprocessor. If the microprocessor cannot be trusted, no security guarantees can be provided by the system. Providing trust in microprocessors, however, is becoming increasingly difficult because of economic, technological and social factors. Increasing use of third-party "soft" intellectual property components, the global scope of the chip design process, increasing processor design complexity and integration, the growing size of processor design teams and the dependence on a relatively small number of designers for a sub-component, all make hardware highly susceptible to malicious design. A sufficiently motivated adversary could introduce backdoors during hardware design. For instance, a hardware designer, by changing only a few lines of Verilog code, can easily modify an on-chip memory system to send data items it receives to a shadow address in addition to the original address. Such backdoors can be used in attacking confidentiality *e.g.,* by exfiltrating sensitive information, integrity *e.g.,* by disabling security checks such as memory protection, and availability *e.g.,* by shutting down the component based on a timer or an external signal. Some recent high-profile attacks have been attributed to untrustworthy microprocessors [10]; hardware trust issues have been a concern for a while now in several domains, including in military and public safety equipment [67], and this issue has attracted media attention lately [45].

Because hardware components (including backdoors) are architecturally positioned at the lowest layer of a computational device, it is very difficult to detect attacks launched or assisted by those components: it is theoretically impossible[2] to do so at a higher layer *e.g.,* at the operating system or application, and there is little functionality available in current processors and motherboards to detect such misbehavior. The state of practice is to ensure that hardware comes from a trusted source and is maintained by trusted personnel — a virtual impossibility given the current design and manufacturing realities. In fact, our inability to catch accidental bugs with traditional design and verification procedures, even in high-volume processors [59], makes it unlikely that hidden backdoors will be caught using the same procedures, as this is an even more challenging task.[3]

In this paper we investigate how microprocessor trust can be strengthened when manufactured via an untrusted design flow. Figure 1 shows the standard steps used to manufacture microprocessors. This paper focuses on one of the initial production steps, which is the coding phase of hardware design (register transfer level, or RTL). Any backdoor introduced during the initial phase becomes progressively more difficult to catch as it percolates through optimizations and tools in the

[1]Appears in *Proceedings of the 31st IEEE Symposium on Security & Privacy* (Oakland), May 2010

Free to distribute for educational use. Copyright restrictions may apply otherwise.

[2]It should be noted, however, that in practice it may be possible to detect discrepancies in the state of the system, such as cache misses. Such detection cannot be guaranteed, and it largely depends on both external artifacts used for the detection (*e.g.,* a reference time source) and on sub-optimal implementation of the backdoor.

[3]The International Technology Roadmap for Semiconductors notes that the number of bugs escaping traditional audit procedures will increase from five to nine per 100,000 lines of code in the coming years [2].

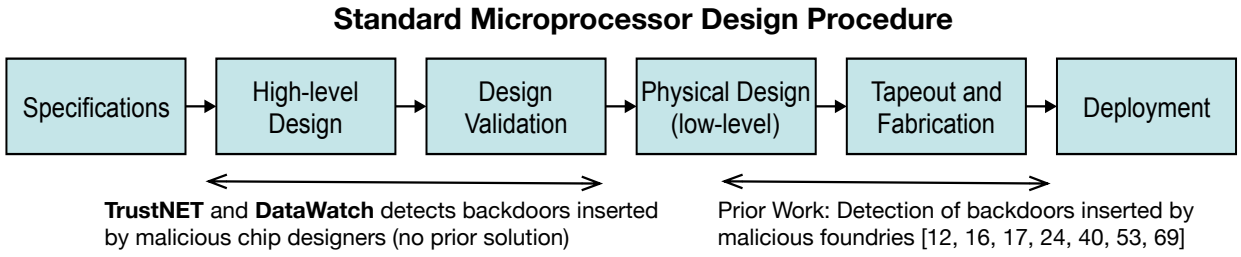

Fig. 1. Microprocessor design flow and scope of this paper.

later phases. Prior work on detecting attacks on hardware by malicious foundries [12][17][16][24][40][53][67] assumes as a starting point the availability of a trusted RTL model, called a golden netlist. Our work aims to provide this trusted, golden netlist.

The traditional approach to building trustworthy systems from untrustworthy components is to redundantly perform a computation on several untrustworthy components and use voting to detect faulty behavior. For example, N processors designed by different designers can run the same instructions, and the most popular output can be accepted. This solution, however, is not viable for microprocessors because it increases the initial design cost significantly by increasing the size of the design team and verification complexity of the design. This solution also increases the recurring operational costs by decreasing performance and increasing power consumption. In this paper, we describe a novel method for building a trustworthy microprocessor (at low cost) from untrusted parts, without the duplication required by the N version model.

Our technique exploits the standard division of work between different sub-components (or units) within a microprocessor, universally available in microprocessor designs. We do this by recognizing simple relationships that must hold between on-chip units. The underlying observation that drives our technique is that the execution of any instruction in a microprocessor consists of a series of separate but tightly coupled microarchitectural events. For example, a memory instruction, in addition to using a cache unit needs to use the fetch, decode and register units. We take advantage of this cooperation in order to detect tampering by noticing that if one unit misbehaves, the entire chain of events is altered.

We explain our technique with an analogy: say, Alice, Bob and Chris are involved in a fund raiser. Alice is the Chief Financial Officer, Chris is a donor, and Bob is a malicious accountant. Let us say Chris makes a donation of $100 towards the fund-raiser and makes the payment to Bob. Let us also say Alice follows all probable donors on Twitter so that she can send a thank you note as soon as donors post tweets on their charitable deeds. Chris tweets: "Donated $100 to charity." Malicious Bob swipes $10 off and reports to Alice that Chris only donated $90. Of course, Alice catches Bob because she can predict Bob's output based on Bob's input from Chris. Applying this analogy to our microprocessor, a malicious cache unit cannot send two outputs when in fact only one memory write instruction has been decoded. Any unit that observes the output of the instruction decoder and output of the cache will be able to tell that tampering has happened along the way.

Our method relies on the fact that cooperating units are not simultaneously lying — a reasonable assumption because high-level design engineers on a microprocessor project are typically responsible for only one or few processor units but not all [26, 46]. Using these relationships, our system, called TRUSTNET, is able to provide resilience against attacks to any *one* unit, even if that unit is a part of TRUSTNET itself. Further, TRUSTNET does not require that any specific unit is trusted. A second system, called DATAWATCH, watches select data on the chip in order to protect against attacks that alter data values without directly changing the number of outputs. Continuing on the previous analogy, this would be a case where Bob, the evil accountant, passed on the full $100, but passed on Canadian dollars instead of American dollars, keeping the difference for himself. When DATAWATCH is active, Chris' tweet would contain the fact that he donated American dollars, tipping off Alice about Bob's crime.

In this paper, we evaluate the resiliency of TRUSTNET and DATAWATCH against a set of attacks implementable in RTL during the initial processor design steps. We show that TRUSTNET and DATAWATCH protect the pipeline and cache memory systems for a microprocessor closely matching the Sun Microsystems' OpenSPARC T2 processor against a large class of attacks at the cost of negligible storage (less than 2 KB per core) and no performance loss. Additionally, TRUSTNET and DATAWATCH, in concert with pre-existing solutions (partial duplication [25]), can provide coverage against many known hardware design level backdoors.

In summary, the primary contributions of this paper are:

- We present a taxonomy describing the attack space for microprocessor designs. The key observation that forms the basis of this taxonomy is that a microprocessor attack can only change the number of instructions or corrupt instructions.
- We present a novel, general solution that exploits the division of work and causal structure of events inherent in microprocessors for detecting a large class of attacks created during the initial stages of microprocessor design by knowledgeable, venal, malicious insiders. To the best of our knowledge, we are the first to propose using violation of co-operation invariants in a microprocessors to detect malicious attacks.

The rest of the paper is organized as follows: Section II describes related work. Section III describes the threat model, assumptions of our study and a taxonomy of attacks. In Section IV we describe our solution. Section V presents

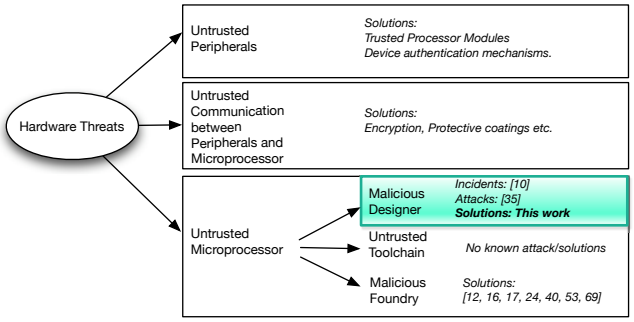

Fig. 2. Proposed work in the context of broader work on hardware threats. Prior countermeasures against hardware threats rely on a trusted microprocessor which this work aims to provide.

evaluation. We conclude and present directions for future research in Section VI.

II. Related Work

Microprocessors are one part of a large ecosystem of hardware parts that forms the trusted computing base. There has been a significant amount of work over the past several decades on protecting different aspects of the ecosystem (Figure 2). In this section, we discuss threats and countermeasures against all classes of hardware, not just microprocessors.

So far hardware, collectively the processor, memory, Network Interface Cards, and other peripheral and communication devices, has been primarily susceptible to two types of attacks: (1) non-invasive side-channel attacks and (2) invasive attacks through external untrusted interfaces/devices. We define an attack as any human action that intentionally causes hardware to deviate from its expected functionality.

Physical side-channel attacks compromise systems by capturing information about program execution by analyzing emanations such as electromagnetic radiation [31, 33, 42, 47, 53] or acoustic signals [15, 44, 60] which occur naturally as a byproduct of computation. These attacks are an instance of covert channels [39] and were initially used to launch attacks against cryptographic algorithms and artifacts (such as "tamper-proof" smartcards [43][37]) but general-purpose processors are also pregnable to such attacks. There have been several attacks that exploit weaknesses in caches [5, 8, 19, 21, 48, 49, 50, 51, 51, 52] and branch prediction [6, 7, 9]. Some countermeasures against these threats include self-destructing keys [32, 35, 62, 72] and new circuit styles that consume the same operational power irrespective of input values [27, 38, 58, 64, 65] and microarchitectural techniques [11, 22, 63, 66, 69].

Invasive untrusted device attacks typically are carried out by knowledgeable insiders who have physical access to the device. These insiders may be able to change the configuration of the hardware causing system malfunction. Examples of such attacks include changing the boot ROM, RAM, Disk or more generally external devices to boot a compromised OS with backdoors or stealing cryptographic keys using unprotected JTAG ports [13][56]. A countermeasure is to store data in encrypted form in untrusted (hardware) entities. Since the '80s there has been significant work in this area [61]. Secure co-processors [28, 35] and Trusted Platform Modules [4] have been used to secure the boot process. More recently, enabled by VLSI advances, researchers have proposed continuous protection of programs and on-chip methods for communication with memory and I/O integration [29, 40].

A new threat that has recently seen a flurry of activity is intentional backdoors in hardware. As hardware development closely resembles software development both in its global scope and liberal use of third party IP, there is growing interest and concern in hardware backdoors and their applications to cyber offense and defense. Broadly speaking, work in this area can fall into one of three categories: threats and countermeasures against malicious designers, threats and countermeasures against malicious design automation tools, and threats and countermeasures against malicious foundries. There has been some work on detecting backdoors inserted by malicious foundries that typically rely on side-channel information such as power for detection [12, 16, 17, 24, 41, 54, 57, 70]. There has been no work on providing countermeasures against malicious designers, which this work aims to address.

There have been a few unconfirmed incidents of design-level hardware attacks [10] and some work in academia on *creating* hardware backdoors. Shamir *et al.* [20] demonstrate how to exploit bugs in the hardware implementation of instructions. King *et al.* [36] propose a malicious circuit that can be embedded inside a general-purpose CPU and can be leveraged by attack software executing on the same system to launch a variety of attacks. They demonstrate a number of such hybrid software/hardware attacks, which operate at a much higher abstraction level than would generally be possible with a hardware-only attack. Although they do not discuss any protection or detection techniques, their work is particularly illuminating in demonstrating the feasibility and ease of creating such attacks through concrete constructs.

III. Threat Model

A malicious hardware designer has to be strategic in creating backdoors because processor development, especially commercial development, is a carefully controlled process. Broadly speaking, the attacker has to follow two steps: first, design a backdoor for an attack, and second, build a trigger for the attack. Just like regular design, the attacker has to handle trade-offs regarding degrees of deception, time to completion, verification complexity, and programmability. In this section we discuss these tradeoffs for attack triggers (Section III-B) and attack backdoors (Section III-C). However, we begin our discussion by detailing assumptions in our threat model.

A. Assumptions

• *Assumption #1: Division of Work* Typically, a microprocessor team is organized into sub-teams, and each sub-team is responsible for a portion of the design (*e.g.,* fetch unit or load-store unit). Microprocessor design is a highly cooperative and structured activity with tens to hundreds of participants [14]. The latest Intel Atom Processor, for instance, is reported to

have had 205 "Functional Unit Blocks" [3]; a design of a recent System-on-Chip product from ST Microelectronics is reported to have required over 200 engineers hierarchically organized into eight units [1]. We assume that *any* sub-unit team in a design can be adversarial but that not more than one of the sub-units can be simultaneously compromised. While adversarial nation-states could possibly buy out complete teams to create undetectable malicious designs, it is more likely that attackers will be a small number of "bad apples."

• *Assumption #2: Access* The focus of this work is to detect the handiwork of malicious microprocessor designers, which includes chip architects, microarchitects, RTL engineers and verifiers, and circuit designers. These workers have approved access to the design, privilege to change the design, and an intricate knowledge of the microprocessor design process and its workings. A malicious designer will be able to provision for the backdoor either during the specification phase, *e.g.,* by allocating "reserved bits" for unnecessary functions, or by changing the RTL. We assume this will be unnoticed during the implementation phase and after the code reviews are complete. Our assumption that code audits will not be able to catch all backdoors is justified because audits are not successful at catching all inadvertent, non-malicious design bugs.

• *Assumption #3: Extent of Changes* The malicious designer is able to insert a backdoor: (i) using only low tens of bits of storage (latches/flops etc.) (ii) with a very small number of logic gates and (iii) without cycle level re-pipelining. This assumption does not restrict the types of attacks allowed. However, we assume the attacker is clever enough to implement the changes in this way. This assumption ensures that the malicious designer can slip in the hardware backdoor unnoticed past traditional audit methods with very high probability.

• *Assumption #4: Triggers* Although an unintentional bug can have the same consequences as a malicious backdoor, a critical difference is that unlike a bug, a backdoor may not be always active. If the backdoor is always active, there is a high chance of detection during random, unit-level design testing. To avoid detection, the malicious designer is likely to carefully control when the backdoor is triggered.

• *Assumption #5: ROMs* We assume that ROMs written during the microprocessor design phase contain correct data. In particular, we assume that microcoded information is correct. The reason for this assumption is that the data in ROMs is statically determined and not altered by the processor's state. For this reason, we consider this security issue to be better solved statically than at runtime.

B. Attack Triggers

An RTL level attacker can use two general strategies for triggering an attack: a time-based trigger or a data-based trigger. From the RTL perspective, input data and the passage of time are the only factors determining the state of the microprocessor (any attack using environmental factors would be a side-channel attack; we are concerned with attacks using digital input signals), so these two strategies or some combination of them are the only ones possible.

• *Trigger #1: Cheat Codes (CC)* A malicious designer can use a sequence of uncommon bits, embedded in either the instruction or data stream to unlock/lock the backdoor. For instance, a store instruction to a specific address and a certain value (one pairing in a 2^{128} space for a 64-bit microprocessor) can be used as a key to unlock a backdoor. Since the search space is so large, the chance that this trigger is hit by random verification is negligible. King *et al.* describe a variant of this attack in which a sequence of instructions in a program unlocks a trigger. The CC method gives an attacker a very high degree of control on the backdoor but may require a reasonably sophisticated state machine to unlock the backdoor. Further, it requires execution of software that may not be possible due to access restrictions. This is due to the fact that in order to ensure the 'magic' instruction(s) is issued, the attacker must execute a program containing that instruction(s). If the attacker cannot obtain access privileges, then this will not be possible.

• *Trigger #2: Ticking Timebomb (TT)* An attacker can build a circuit to turn on the backdoor after the machine has been powered on for a certain number of cycles. The TT method is very simple to implement in terms of hardware; for instance, a simple 40-bit counter that increments once per processor clock cycle can be used to open a backdoor after roughly 18 minutes of uptime at 1 GHz. Unlike the CC method, TT triggers do not require any special software to open the backdoor. However, like CC triggers, TT triggers can easily escape detection during design validation because random tests are typically not longer than millions of cycles.

C. Backdoor Types

While the space of possible attacks is limited only by the attacker's creativity and access to the design, attacks can be broadly classified into two categories, based on their runtime characteristics. We observe that an attacker can either create a hardware backdoor to do more (or less) work than the uncompromised design would, or he/she can create a backdoor to do the same amount of work (but work that is different from that of an uncompromised unit). By work, we mean the microarchitectural sub-operations or communications that must be carried out for the execution of an instruction. This is a complete, binary classification.

• *Emitter Backdoors (EB)* An emitter backdoor in a microarchitectural unit explicitly sends a different number of microarchitectural communication than an uncompromised unit. An example of an emitter backdoor in a memory unit is one that sends out loads or stores to a shadow address. When this type of attack is triggered, each memory instruction, upon accessing the cache subunit, sends out two or more microarchitectural transactions to downstream memory units in the hierarchy. Similar attacks can also be orchestrated for southbridge (I/O control hub) components, such as DMA and VGA controllers, or other third party IP, to exfiltrate

confidential data to unauthorized locations.

• *Corrupter Backdoors (CB)* In this type of attack, the attacker changes the results of a microarchitectural operation without directly changing the number of microarchitectural transactions. We consider two types of corrupter backdoors — control corrupters and data corrupters.

A control corrupter backdoor alters the type or semantics of an instruction in flight in a way that changes the number of microarchitectural transactions somewhere else on-chip (*e.g.,* at a later cycle). These attacks are similar to emitter attacks, except that instead of simply issuing an extra instruction, they use some part of a legitimate instruction in order to change the number of transactions happening on-chip. For example, if a decode unit translates a no op instruction into a store instruction, this will indirectly cause the cache unit to do more work than it would in an untampered microprocessor. However, this change will not manifest itself until a later cycle. This is different from an emitter attack because the decode unit does not insert any new transactions directly; it decodes exactly the same number of instructions in the tampered and untampered case, but the value it outputs in the tampered case causes the cache unit to do more work a few cycles later.

Data corrupter backdoors alter only the data being used in microarchitectural transactions, without in any way altering the number of events happening on-chip during the life of the instruction. Examples of this could include changing the value being written to a register file or changing the address on a store request. For instance, an instruction might be maliciously decoded to turn an addition into a subtraction, causing the ALU to produce a difference value instead of a sum value. [4]

• *Emitter vs. Corrupter Trade-offs* From the attacker's point of view, emitter attacks are easy to implement. Emitter attacks, such as shadow loads, have very low area and logic requirements. They also have the nice property (for the attacker) that a user may not see any symptoms of hardware emitters when using applications. This is because they can preserve the original instruction stream. Often in prior work the term 'backdoor' actually means 'emitter backdoor.'

Corrupter attacks, on the other hand, are more complicated to design and harder to hide from the user. In fact, a control corrupter attack requires strictly more logic than a similar emitter attack because rather then simply sending a trigger, it must hide the trigger within a live instruction (which involves extra multiplexing or something equivalent). In these attacks, rather than simply emitting bogus signals, the user's own instructions are altered to invoke the attack. Since the user's instructions are being altered, the attacker must have some knowledge of the binaries being run to change the data without tipping off the user. If the execution of the backdoor caused the user's program to crash, this would violate the secrecy of the attack. Corrupter attacks also scale poorly with datapath sizes, since they require decoding of user instructions. In the case of multi-stage decoders, the backdoor itself may require latches and execute over multiple cycles.

To summarize, the "biggest bang for the buck" for the attacker is from ticking-timebomb-emitter attacks. They can be implemented with very little logic, are not dependent on software or instruction sequences, and can run to completion unnoticed by users. In the following section, however, we discuss strategies for defending against all types of backdoors and triggers.

[4]Data corrupter backdoors can be used to change program flow, for example by changing a value in a register, thus changing the result of a future 'branch-if-equal' instruction. However, each individual instruction will still do the same amount of work as it should. The extra work will not occur until the corrupt instruction has been committed. Thus each instruction considered individually will appear to be doing the correct amount of work.

IV. Principles for Microprocessor Protection

We propose as a solution to the untrusted designer problem an on-chip monitoring system that recognizes malicious behavior at runtime, regardless of the trigger or unit. Different attacks require different defenses. As such, we present our solution in four flavors. We first describe low overhead solutions for emitter and control corrupter protection, called **TrustNet** and **DataWatch**. We then describe how a form of partial duplication, which we call 'smart duplication' can be used against some data corrupters. For data corrupters not protected by any of the above mechanisms, we recommend full duplication. For this initial study, we discuss our solutions in the context of simple microprocessors that do not re-order instructions.

A. Emitter Backdoor Protection

Emitter backdoors by definition cause more (or less) microarchitectural transactions to occur in the corrupted unit than the instruction specifies. We designed the **TrustNet** monitoring system to watch the microarchitectural transactions in each unit and catch this class of attacks. Conceptually, the system detects violations of deterministic communication invariants between on-chip units, which are violated by emitter backdoors.

Toward this end, we designed the prediction/reaction monitor triangle, depicted in Figure 3. A triangle consists of three different on-chip units - a predictor, a reactor, and a target (monitored unit in Figure 3). The predictor unit sends messages to the monitor, predicting events that should come out of the target unit. If the reactor does not receive a predicted

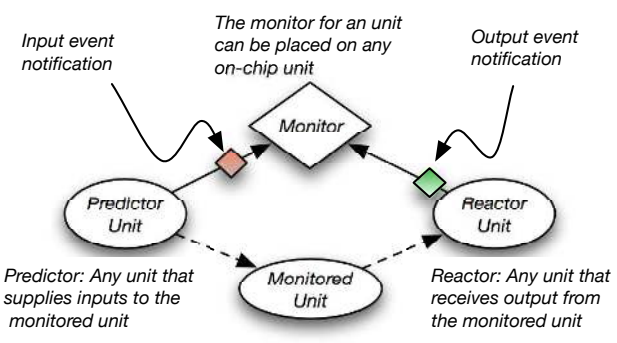

Fig. 3. Overview of the **TrustNet** and **DataWatch** monitoring scheme.

event, or if the reactor receives an event that was never predicted, then the monitor throws an alarm.

The target unit is any unit on-chip. For one example, using Figure 4 as reference, consider the decode unit (IDU) as a target. The predictor unit can be any unit that sees events before they reach the decode unit, for example the fetch unit (IFU). The fetch unit sees every instruction before it reaches the decode unit. The reactor unit can be any unit that sees events after they pass through the target. For example, it can be the execution unit (EXU), because that unit always receives information about instructions after they pass through the decode unit. The monitor itself can reside anywhere on-chip. The IFU knows each cycle whether or not it has fetched a new instruction. The EXU knows at each cycle whether or not a valid instruction is advancing through the pipeline. Having these two sources of knowledge corroborate prevents the IDU from ever inserting bogus instructions into the pipeline. Nothing extra has to be computed, since this knowledge is already represented by signals in the respective units.

A vital aspect of this design is that the target unit never communicates with the monitor. Therefore, even if the designer of unit X knows the design of the monitor (which is likely), the designer is unable to corrupt the monitor of X. For this reason, it is important that the monitor of X should not be physically part of unit X.

Recall the assumption from Section III that only one subteam is corrupt, and so an attacker cannot corrupt two independent units on-chip. This assumption guarantees that our system is secure against the attack space. Consider the monitor set up to watch some unit X. There are four items in play - the predictor of X, which we will call P; the reactor to X, which we will call R; X itself; and the monitor of X, which we will call M. The attacker must choose one and only one of these items to corrupt. In order to cause a confidentiality or integrity attack in X, the attacker would have to choose to corrupt X itself. However, in that case P, R, and M are all untampered with, and the attack is caught at runtime by the P-R-M monitor. The attacker could instead tamper with P or R (but not both), but in either case that attack would be caught at runtime, since P and R would disagree with each other. The last choice is to corrupt M to cause an availability attack. However, since M contains only one (or a few) logic gates, it is too small to contain a cheat code or ticking time bomb. M has so few states (usually none) and input bits (usually two) that it can easily be verified statically by exhaustive enumeration.

B. Control Corrupter Backdoor Protection

Recall that control corrupters change the semantics of an instruction in flight. The same predictor/reactor scheme, used to catch emitter attacks, works to detect control corrupter attacks, since these attacks either cause reactions (microarchitectural transactions) that are not predicted or prevent transactions that are predicted. To this end, we present **DataWatch**, the natural extension of **TrustNet**. **DataWatch** uses the same triangle scheme as **TrustNet**. However, instead of transmitting individual bits to monitors, signatures (*e.g.,* hashes, parity bits, slices) are passed to monitors that check for equivalence. For example, in the case of the decoder, a two bit signature is enough to define whether an instruction involves a load, a store, both, or neither (two bits for four cases). An equality check on the signatures from the predictor and the reactor is enough to confirm an on-chip invariant, namely that the number of each type of regular memory operation that enters the IDU equals the number of each type of memory operation that exit the IDU. This prevents, for instance, an attack that exfiltrates data by converting no ops into store instructions.

TABLE I
COMPARISON OF **TrustNet**, **DataWatch**, AND SMART DUPLICATION FOR SIMPLE, IN-ORDER MICROPROCESSORS

Attribute	TrustNet	DataWatch	Smart Duplic.
Data Corrupter Detection	No	No	Yes
Control Corrupter Detection	No	Yes	Yes
Emitter Detection	Yes	Yes	Yes
False Positives	No	No	No
Stalls Processor	No	No	Possibly
Area Cost	Negligible	Low	Moderate
On-Chip Traffic	Increases		
Off-chip memory	No effect		
Backdoor source	Can be identified		

Data corrupter attacks change only the data that is sent in on-chip communications. These backdoors are fundamentally different from the types previously discussed because the amounts and types of communications between units during the execution of an instruction is identical to that of a correctly functioning chip. The monitor triangle, while very efficient for recognizing amounts and types of transactions, does not work well for this case, because data corrupter attacks cannot be recognized without duplicating some of the computational logic that has been corrupted. For example, if the EXU (execution unit) produces an incorrect sum, the fact that the sum is wrong cannot be known without duplicating (or otherwise performing the job of) the ALU (arithmetic/logic unit).

However, this type of attack has some similarities with transient errors that can occur in microprocessors. Significant work has been done toward transient error detection [25][55][71][23] and fault tolerance, and we draw on the principles of some of this prior work. It is sufficient in many cases to duplicate select computational logic in order to protect the RTL design, since standard memory structures (*e.g.,* RAMs) are not susceptible to RTL level attacks. We propose that this type of minimal duplication, which we call 'smart duplication,' can be used in a case-by-case way to protect any units (*e.g.,* memory control unit) that are not covered by the **DataWatch** system or any units that may be considered vulnerable to data corrupter attacks. This partial duplication allows for protection against data corrupter attacks. However, it does this at the possible cost of processor stalls and extra area, and as explained previously(Sec. III-C), in most domains data corrupter attacks would likely be considered infeasible due to the requisite of knowing the binaries that will be run in the future during the RTL design phase. Therefore, this

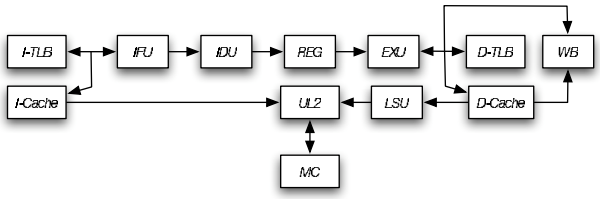

Fig. 4. Units and communication in the hypothetical inorder processor used in this study.

technique may only be useful in a few select domains or not at all.

Table I summarizes some of the attributes of the offered solutions. None of the proposed solutions have a problem with false positives (false alarms) because they use invariants that can be easily determined statically in non-speculative, in-order microprocessors. Extending this solution to designs with advanced speculative techniques, such as prefetching, may make false positive avoidance non-trivial. False negatives (missed attacks) are only a problem if multiple signals in the **DATAWATCH** technique are hashed to save space, because two different values may hash to the same key, thus tricking the equality checker. However, hashing is an implementation option, which we chose to avoid because the space requirement of the baseline **DATAWATCH** system is fairly low.

C. A Case Study

To demonstrate the principles of the **TRUSTNET** and **DATAWATCH** techniques we describe how they can be applied to a hypothetical non-speculative, in-order microprocessor. The in-order microprocessor used in this study closely models the cores and cache hierarchy of the OpenSPARC T2 microprocessor with the exception of the cross bar network between core and memory system, the thread switching unit, and the chip system units such as the clock and test units. For this study, the units in the processor core are partitioned as described in the OpenSPARC T2 documentation and we used the open source RTL code to identify the predictors and reactors for each unit. The following are the **TRUSTNET** monitoring triangles we implemented, categorized by the unit being monitored:

- *#1 IDU:* The primary responsibility of the IDU is to decode instructions. Predicted by the IFU and reacted to by the EXU, the IDU monitor confirms each cycle that a valid instruction comes out of the IDU if and only if a valid instruction entered the IDU. This monitor detects any attack wherein the IDU inserts spurious instructions into the stream. In the case of branch and jump instructions, which do not go all the way through the pipeline, the information travels far enough for the EXU to know that a branch or jump is occurring. This monitor can be extended to support a speculative microprocessor if the monitor can reliably identify speculative instructions.
- *#2 IFU:* The primary responsibility of the IFU is to fetch instructions. Predicted by the I-Cache and reacted to by the IDU, this monitor confirms each cycle that a valid instruction comes out of the IFU if and only if an instruction was fetched from the I-Cache. This invariant catches any attack wherein the IFU sneaks instructions into the stream that did not come from the I-Cache. The monitor operates on the level of single instructions as opposed to whole cache lines. While the whole line is loaded into the I-Cache from the L2, the I-Cache knows when individual instructions are being fetched into the IFU.
- *#3 LSU:* The load-store unit (LSU) handles memory references between the SPARC core, the L1 data cache and the L2 cache. Predicted by the IDU and reacted to by the D-Cache, this monitor confirms each cycle that a memory action (load or store) is requested if and only if a memory instruction was fed into the LSU. This catches shadow load or shadow store attacks in the LSU. Our microprocessor uses write merging, which could have been a problem, since several incoming write requests are merged into a single outgoing write request. However, there is still a signal each cycle stating whether or not a load/store is being initiated, so even if several writes are merged over several cycles, there is still a signal each cycle for the monitoring system.
- *#4 I-Cache:* Predicted by the IFU and reacted to by the unified L2 Cache, this confirms each cycle that an L2 instruction load request is received in the L2 Cache if and only if that load corresponds to a fetch that missed in the I-Cache. The IFU can predict this because it receives an 'invalid' signal from the I-Cache on a miss. An I-Cache miss immediately triggers an L2 request and stalls the IFU, so there is no issue with cache line size. The IFU buffers this prediction until the reaction is received from the L2 Cache. This catches shadow instruction loads in the I-Cache.
- *#5 D-Cache:* Predicted by the LSU and reacted to by the L2 Cache, this is the same as the monitor #4 but watches data requests instead of instruction requests.
- *#6 L2 Cache:* Predicted by the I-Cache and reacted to by MMU, this is the same as monitor #4 but is one level higher in the cache hierarchy.
- *#7 L2 Cache:* Predicted by the D-Cache and reacted to by the MMU, this is the same as monitor #5 but is one level higher in the cache hierarchy.
- *#8 D-Cache:* Predicted by the LSU and reacted to by the L2 Cache, this is the same as monitor #5 but watches writes instead of reads. It is necessary that two separate monitors watch reads and writes; if a single monitor counted only the total number of reads and writes, then an attacker could convert a write into a read unnoticed. This would cause old data to be loaded into the cache and prevent the new value from being written.
- *#9 L2 Cache:* Predicted by the D-Cache and I-Cache and reacted to by the MMU, this confirms that line accesses in the MMU correspond to line accesses issued by the level 1 caches. This monitor prevents shadow loads/stores executed by the L2 Cache.

The following are the **DATAWATCH** monitoring triangles we implemented, categorized by the unit being monitored:

- *#10 IFU:* Predicted by the IDU and reacted to by the I-

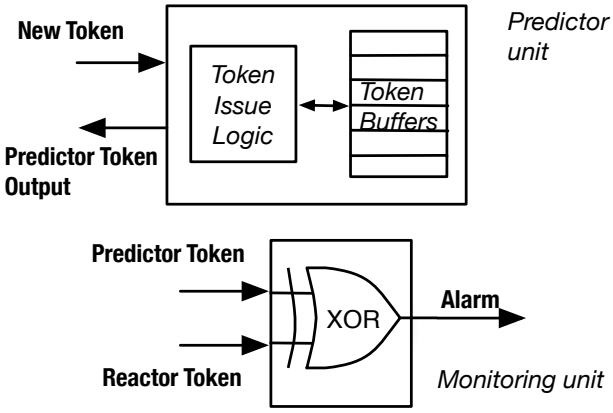

Fig. 5. **TRUSTNET** Monitor Microachitecture.

Cache, this confirms each cycle that if the I-Cache receives a valid PC value it is the same as the value computed in the IFU. This required some duplication of PC logic but did not require any extra storage. This prevents attacks wherein the IFU maliciously changes the control flow.

• *#11 Data TLB:* Predicted by the checker data TLB and reacted to by the LSU, this confirms each cycle that the outputs of the data TLB match the outputs of the checker data TLB. This prevents data TLB attacks, such as permissions violations or page mis-translation. This is on the borderline of what we would start to call 'smart duplication' because the 'signatures' are so large. However, we included this is our **DATAWATCH** implementation and simulation.

• *#12 Instruction TLB:* Predicted by the checker instruction TLB and reacted to by the IFU, this is the same as monitor #11 but for the instruction TLB rather than the data TLB.

• *#13 IDU:* Predicted by the IFU and reacted to by the LSU, this confirms that the number of instructions decoded into memory operations matches the number of memory instructions fetched. For our microprocessor, this required that the IFU looked at a few bits of the instruction. The monitoring occurs at a one cycle lag, so the timing on the critical path is unaffected. The IFU stores a few of the bits from the fetched instruction in flip-flops until the next cycle, when a prediction can be made with a few logical gates. For our case study, this is the only type of control corrupter decoder attack we address. The reason for this is that in our simple microprocessor, the only types of signals the decoder can cause are loads in stores (if, for example, the decoder changed an add to a subtract, this would be a data corrupter, because it would not alter the number of transactions in the execution unit, just the value of the output). In more complex microprocessors, decode units may be responsible for more types of transactions and might require additional monitoring triangles. When customizing a **DATAWATCH** system to fit a particular design, it is important up front to identify what types of signals each unit is responsible for.

D. Microarchitecture and Optimizations

The microarchitecture of the predictor and monitor units are depicted in Figure 5. The predictor unit consists of (i) event buffers for delaying the issue of tokens to the monitor and (ii) token issue logic to determine when buffered events can be released from the event buffers. The predictor unit requires a small buffer because it is possible for multiple predictions to happen before a reaction happens, and these predictions must be remembered for that duration. These buffers can be sized *a priori* to avoid overflows. The monitor itself simply checks if events appear on the predictor and reactor inputs during the same cycle.

1) **TRUSTNET** *Optimization:* When designing the **TRUSTNET** system to catch emitter backdoors, we considered it to be important that the monitors fit simply into the pipeline without any complex timing or buffering issues.

Since predictions and reactions must arrive at the monitor during the same cycle, timing must be controlled in the face of non-determinism, which arises in all microprocessors due to cache misses, etc. We handled this differently in the case of the memory hierarchy and in the case of the pipeline. The pipeline offers a natural lock-step manner for coordinating events. If a reaction stage is N pipeline steps down from a prediction stage, then the prediction stage has a size N buffer that advances only when that stage of the pipeline advances. Since the monitoring network advances in lock-step with pipelined events, timing is not a problem. For example, if the third pipeline stage wants to send a prediction to a monitor that lies in the fifth pipeline stage, this will take two pipeline advancements (no need for forwarding). If the third stage stalls for any reason, the prediction also stalls and gets buffered. When the data from the third stage reaches the fifth stage, the prediction token will also arrive. Of course, the prediction token should not pass through the fourth stage but should instead remain in the prediction buffer, with a bit denoting that it is semantically in the fourth stage.

In the case of the cache hierarchy, on the other hand, it is necessary to know which predictions correspond to which reactions, because it is possible for memory requests to be handled out of order. This requires time-stamping of packets, for example with a one byte local time signature copied from an 8-bit modular counter.

2) **DATAWATCH***Optimization:* A naíve solution for catching control corrupter backdoors in TLBs (translation lookaside buffers) is to simply have two (or more) designers design the same TLB and compare their outputs each cycle. Since TLBs tend to be power-hungry, highly associative structures, duplication is not a good idea. Instead of complete duplication, we propose a new TLB microarchitecture that provides significant protection without the costs associated with duplication. The TLBs contain page translation and permissions information not available elsewhere on chip. A TLB consists of a CAM that translates a virtual page into a physical page, which is then stored in a table (RAM) with the corresponding permissions information for that physical page.

The basic idea of our method is to create a "checker" direct-mapped structure that has the same functionality as a TLB, the motivation being that a direct-mapped structure uses a fraction of the power of an associative one. The TLBs in our case study are fully associative. We added functionality to the CAMs to output the line number of the output. This allowed us to build a checker TLB that uses these line numbers. Essentially, instead of having one CAM and a direct-mapped RAM (as is normal), we have one CAM and two direct-mapped RAMs that operate in parallel. The CAM provides matching entries to both RAMs in parallel. One of those RAMs communicates with the rest of the chip while the other RAM only gives outputs to a monitor (equality verifier). The equality check occurs at a one cycle latency, so the values are buffered for that cycle.

Naturally, the CAM could be tampered with so that it sends incorrect line numbers to the checker TLB. This would cause the equality check to fail because data from one line of the original TLB's RAM will be compared to data from a different line of the second RAM, causing an alarm to be thrown. Therefore, our checker TLB turns a potential confidentiality or integrity attack into at worst an availability attack. We note that this availability attack would also be easy to catch at verification time because the passing of the line number is simple, combinatorial logic that can be checked by exhaustive enumeration.

While this duplication is much more expensive than the simpler monitor used for emitter backdoor protection, it is much less expensive than complete duplication and offers strong protection for a highly vulnerable unit.

E. Applications of Prior Solutions

As we mentioned briefly in the introduction, the problem of building trusted systems from untrustworthy components is a classic problem that has received some attention in the systems community. A common solution used to amplify trust in corruptible processes is to use the N-version model of computation. The basic idea is to have N entities perform the same computation and compare the N outputs to check for untrustworthy behavior. In this section, we expand on the different ways in which this concept can be applied to microprocessors and discuss the advantanges and disadvantages.

To deal with untrusted designers in the context of microprocessors, one option is to have N designers create N versions of each unit within a processor, which would all be run continuously to check for untrustworthy behavior. Alternately, one could run a program on N different systems that implement the same ISA but are manufactured by different vendors, say, boards that have x86 processors from AMD, Intel and Centaur. The latter suffers from high power overhead while the former suffers from both high design cost per chip and high runtime costs. Another solution that avoids only the runtime cost is to statically and formally check the design units from N designers for equivalence. This approach increases the design cost and does not scale to large designs or designs that are vastly different. According to the 2007 ITRS roadmap, only 13.8% of a normal microprocessor design specification is formalized for verifiability [2]. All common solutions to this problem appear unsatisfactory in the context of microprocessors.

Another option is to use static verification to identify backdoors. There has been extensive prior work on static verification of RTL level designs [68][18][34]. Static verification involves confirming functional equivalence between a behavioral level golden model (*e.g.,* a C program) and the RTL level design under test. The difficulty lies in the fact that the input space for a microprocessor grows exponentially with the number of input interfaces and the internal state size, which makes the functional domain catastrophically large. Exhaustive comparison is unrealistic, so the state of the art is to use probabilistic approaches that attempt to obtain reasonable coverage, such as equivalence checking [30][68], model checking [30], and theorem proving [30]. These approaches can work for small units, particularly ones with little or no state, such as ALUs. Unfortunately, static verification is increasingly becoming the bottleneck in the microprocessor design process [30] and is becoming less reliable [2].

A fundamental weakness of static verification techniques when it comes to backdoor detection is that they attempt to use a stationary weapon to hit a moving target. Static methods choose specific targets for comparison or invariants to confirm about small portions of the design. Since it is reasonable to assume that a malicious insider would have full knowledge of the static verification technique being used, he or she would most likely design the backdoor to avoid the space covered by these techniques. For example, he or she would likely make sure not to violate any of the theorems being verified and to avoid regions being formally checked for equivalence.

V. Evaluation

The goals of our evaluation were to: (1) study the accuracy and coverage provided by **TrustNet** and **DataWatch**, (2) measure the increases in on-chip network congestion from **DataWatch** running on real programs and (3) measure the area overheads of both mechanisms. We do not discuss performance since the proposed mechanisms do not stall the pipeline, memory system, or any other on-chip unit, and security packets travel on a dedicated network.

A. Applicability

This section addresses the general applicability and limitations of our solution, including related aspects and potential extensions.

- *Scope of our solution* Our implementation of **TrustNet** and **DataWatch** was designed for a simple, in-order microprocessor. While the methodology is applicable to any in-order microprocessor, this exact implementation only works for the microprocessor in our case study. In order to fit **TrustNet** and **DataWatch** to other designs, it is necessary to analyze the units at a high level and determine what the natural predictors and reactors are. In future work, we hope to develop a tool that automates this process.
- *Level of our solution* Our solution is at the RTL level and

thus can only catch attacks that operate on the RTL level. Post-RTL, circuit level attacks, such as tampering with the voltage thresholds on certain transistors, would not be caught by our system. Our solution covers the cores and the cache hierarchy of the OpenSPARC T2 microprocessor but does not cover debug/test logic or miscellanies, such as clock distribution. Additionally, side-channel attacks are also not covered.

• *Multiple attackers* The solution we implemented works only under the assumption that at most one of the design teams is corrupt. Our design is a triangle (complete graph of size three). If we remove the assumption that only one of the design subteams is corrupt and allow for n different subteams to be corrupt and fully coordinated, then the solution must be come more complicated.

In order for a **TRUSTNET** system to catch n coordinated attackers, it is necessary to form a complete graph of size $n + 2$. The premise of the system is that two honest units must communicate with each other to compare information and detect discrepancies. If there are at most $n+1$ nodes and n have been tampered with, then at most one of them is honest, and the one honest node receives only invalid information. Therefore, the size of the graph must be at least $n + 2$ so that there are at least two honest nodes. If the graph is not complete, *i.e.* it is missing an edge, then it is possible that the missing edge connects the only two honest nodes. In that case, the two honest nodes receive only invalid information. Therefore, the graph must be complete. Since complete graphs contain $\frac{n(n+1)}{2}$ bidirectional edges, the **TRUSTNET** solution, when extended to n attackers for a microprocessor with u units, has a fundamental communication overhead of $n(n + 1)u \propto n^2u$.

The conclusion is that even though **TRUSTNET** and **DATAWATCH** are generalizable for multiple, coordinated attacking subteams, they do not scale well. We present this generalized scheme only for completeness.

• *Alarms* The decision of how to handle an alarm is domain specific and not a fundamental aspect of our monitoring system. However, we present initial suggestive thoughts on what might be done with an alarm. In our experimental implementation, the alarm was simply recorded and not used for any corrective actions.

The simplest response to an alarm is to kill the defective processor, which turns a confidentiality or integrity attack into an availability attack. In highly secure domains, this may be desirable to guarantee no exfiltration of sensitive data. Additionally, in a heterogeneous processor (diversity) setting, it may be desirable to kill the defective core. We also note that using the **TRUSTNET** and **DATAWATCH** infrastructure has the potential to greatly simplify the task of making microprocessors tamper corrective. If an alarm is sounded, the problem can be corrected by rolling back to the last committed instruction. Additionally, the instruction that was in flight in the corrupted unit can be flagged as a cheat code and logged for future execution. This approach would be analogous to a honeypot.

• *Extensions to General Microprocessors* There are several ways to generalize the **TRUSTNET** and **DATAWATCH** architecture, and each way poses challenges for future work. The multi-threaded case is a relatively simple generalization that can be implemented by making the packets n-wide for an n-threaded core. Assuming one thread is not supposed to alter the microarchitectural transactions of another thread, the n-wide packet can function semantically as n independent monitors. The out-of-order case is more complicated as it requires our mechanisms to be extended to handle reordering of in-flight predictor/reactor tokens. Handling speculative techniques would also require extensions, though we believe that the principles of our system can be applied to work in this case without any false alarms by identifying what the lifetime of an instruction is (whether it is prefetched, speculated or committed) and monitoring it for that lifetime. There are other advanced features of modern microprocessors, and each may warrant its own attention in future work. For example, some microprocessors have a privileged or supervisor state that is separate from the permissions governed by the TLB. Such additions would open the door for control corrupter attacks and would warrant additional monitoring triangles.

TABLE II
EXPERIMENTAL INFRASTRUCTURE

Instruction Set	Sun SPARC
Microarchitecture	
Instruction supply	16KB, 8-way 1R/1W L1 I cache, 64-entry FA I-TLB (both 2-cycle access, 52 cycles on TLB miss), No branch prediction, stall until branch resolution.
Execution	Single issue, 1 INT ALU, T2 SPARC register windows.
Data supply	8KB, 4-way L1 D cache 1RW, 128-entry FA DTLB (both 3 cycle access, 53 cycles on TLB miss, write-back policy), unified 4 MB, 16-way L2 cache, 1 RW (both 12 cycle access, write-back policy), Unlimited main memory at 250 cycle access latency.
Pipeline Stages	Fetch, Cache, Pick, Decode, Execute, Read, Bypass, Writeback.
Benchmarks	bzip2, gcc, mcf, gobmk, hmmer, test inputs, base compiler optimizations, SPARC compiler

B. Evaluation Methodology

We demonstrate our design on a simplified model of Sun Microsystems' OpenSPARC T2 microarchitecture. We chose this architecture and instantiation because it is the only "industrial-strength" hardware design that is also available as open source. While our experiments and analysis were performed on our simulated core, based on the OpenSPARC T2 microprocessor design, we use nothing unique to that design, and we believe our techniques can in principle be applied to any microprocessor that has memory hierarchy and pipelines. In our case study, we used the RTL hardware implementation (1) to construct well-formed, meaningful attacks to test the resiliency of the system and (2) to systematically determine the number of on-chip units that can be covered by our design. In addition, to measure congestion, similar to many computer architecture studies, we use a cycle-accurate simulator that exactly models one core of our microprocessor. The details

of our simulation infrastructure are summarized in Table II. We implemented all the TRUSTNET and DATAWATCH monitor triangles discussed in this paper (Tables III, IV) including the partially duplicated TLBs.

C. Attack Space Coverage

To determine how good TRUSTNET and DATAWATCH are at protecting against attacks on microprocessors, we first need to measure the microprocessor attack/vulnerability space. To measure the attack/vulnerability space, we observe that an on-chip unit is only vulnerable to backdoors in-so-far as its interfaces are threatened. What goes on inside the unit doesn't matter so long as everything that goes in and out of it is correct. If all inputs and outputs are the same as in an uncorrupted chip, then there is no problem, because there has been no corruption or exfiltration of data. Therefore, to identify the points of vulnerability, we record the interfaces between on-chip units. The efficacy of our solution is then determined by whether or not these interfaces are protected from attacks using TRUSTNETand DATAWATCH.

Figure 6 (A,B,C,D) shows the distribution of shared interfaces between units within the overall chip, the processor core, the memory elements and system elements respectively, in the RTL implementation of the OpenSPARC T2. Each measurement in the grid represents the number of signals that enter and leave a unit within the processor. All communication is normalized to the unit that has the highest communication. Very small bars (below 0.1) signify connections that are not part of instruction processing, but rather contain miscellaneous information, such as power settings, clock distribution, etc (attacks on these may be possible at fabrication, *e.g.,* misclocking a certain unit, but would be diffcult at the RTL code level). Most of these miscellaneous signals are not present in our case study. Thus, the microprocessor in our case study is somewhat simpler than a real OpenSPARC T2 and lacks some of the less central features, such as power management and floating point operations.

As can be seen from Tables III and IV, that describe known emitter and corrupter monitors respectively, and Figure 6, that describes all the interfaces, that all of the units in the core that have significant communications (more than 0.1) are monitored by TRUSTNET and DATAWATCH. For this study, we used manual analysis to identify which signals on the interfaces can be altered to cause emitter and control corrupter attacks to verify that TRUSTNET and DATAWATCH can cover these signals. Based on this analysis, most of the vulnerable interfaces can be protected against the known attacks. However, our manual analysis may have failed to exhaust all possible attack scenarios; in future work, we hope to automate this process to guarantee full coverage.

While TRUSTNET and DATAWATCH cover the processor cores and the cache hierarchy against the emitter and control corrupter attacks we identified, we did not cover the floating point logic, memory controller units, crossbars and the miscellaneous logic. We believe that formal verification can be used to cover floating point logic as is already in vogue. Similarly, we beleive that the crossbar in our case study is simple enough to be amenable to formal verification. We are not sure if our strategy can cover the miscellaneous logic such as power management unit and the stream processing unit (we have not had time to understand the implementation) but these units have very little contact with other units on chip (less than 0.1 normalized), and we believe it is unlikely that they will significantly increase attack vulnerability. The difficulty with covering the MCU with TRUSTNET is that the MCU communicates with only the main memory and that there is nothing on-chip that can serve as a reactor. The simplest way to handle this special case would be to duplicate the MCU control logic, which serves the same purpose as complete duplication but without increasing memory bandwidth.

D. Attacks and Detection

Based on our study of the OpenSPARC T2 RTL, we constructed the emitter and control corrupter attacks described in Table V and injected the attacks into our simulator to measure the percentage of false negatives and false positives. We did not implement the data corrupter attacks. These attacks, such as the one which alters the address of an otherwise valid load, are situations where some logic duplication may be required. In this case, the address (or a hash of the address) could be forwarded to make sure it has not been altered. However, this was not done in our implementation, which protected only against emitter and control corrupter attacks. As stated earlier, we also did not take any corrective or rollback measures with alarms; we only recorded them.

As was expected, all emitter and control corrupter attacks were caught in all cases. This is very important because it demonstrates that our system provides coverage for all of the units we applied it to and for various types of attacks. We also measured the overall accuracy of our solution with no attacks, as measured by the percentage of the cycles in which there are no false positives thrown. For all tests run, no false positives occurred. It is vital that there are no false positives *and* no false negatives because the latter would be a breach of security and the former would cripple the system.

E. Traffic

Since TRUSTNET and DATAWATCH do not stall the pipeline or otherwise increase computational cycles, the most relevant cost of the system is the increase in on-chip network traffic. This increase depends on the architecture, but it can be bounded in general if we assume a cache hierarchy and one or more pipelined computational units. The total amount of traffic in the worst case is bounded above as per the following equation:

$$\begin{aligned}\text{traffic} \leq 2 * (&\text{MemoryOps} * \text{MemoryMonitors} \\ &+ \text{Instructions} * \text{PipelineMonitors})\end{aligned}$$

The factor of two comes from the fact that each monitoring event consists of two packets - a prediction and a reaction. This is a loose upper bound, and we expect real programs to produce far less than this much traffic. However, this upper

TABLE III
DESCRIPTIONS OF THE EMITTER PROTECTION MONITORS FOR OUR IMPLEMENTATION

Monitored Unit	Predictor	Reactor	Invariant	Example of attack thwarted
IDU	IFU	EXU	# of instructions in = # of instructions out	IDU stalls the fetch unit and sends malicious commands to the EXU
IFU	I-Cache	IDU	# of instructions in = # of instructions out	IFU sends spurious instructions to the IDU
LSU	IDU	D-Cache	# of Memory ops issued = # of Memory ops performed	LSU performs shadow loads
I-Cache	IFU	L2 Cache	# of requested L2 instructions = # of IFU requests that miss	I-Cache returns spurious instructions to IFU while waiting on the L2 Cache
L2 Cache	I-Cache	MMU	# of requested instructions from memory = # of I-Cache requests that miss in L2	L2 Cache returns spurious instructions while waiting on main memory
D-Cache	LSU	L2 Cache	# of requested L2 data = # of LSU requests that miss	D-Cache returns fake data while waiting on the L2 Cache
L2 Cache	D-Cache	MMU	# of requested data from memory = # of D-Cache requests that miss in L2	L2 Cache returns spurious data while waiting on main memory
D-Cache	LSU	L2 Cache	# of L2 cache lines written = # of LSU line writes issued	D-Cache sends write to L2 cache unprompted
L2 Cache	D-Cache	MMU	# of Memory lines written = # of D-Cache line writes issued	L2 sends write to memory unprompted
Legend: IDU = decode unit, IFU = fetch unit, LSU = load/store unit, I-Cache = instruction cache, D-Cache = data cache, L2 Cache = unified L2 cache				

TABLE IV
CORRUPTER PROTECTION MONITORS

Monitored Unit	Predictor	Reactor	Invariant	Example of attack thwarted	Type of signature
IFU	IDU	I-Cache	PC received = PC computed	IFU branches incorrectly	Eight bit signature
D-TLB	Checker D-TLB	LSU	TLB output = checker TLB output	TLB violates permissions	Full permissions and translation
I-TLB	Checker I-TLB	IFU	TLB output = checker TLB output	TLB violates permissions	Full permissions and translation
IDU	IFU	LSU	Memory ops issued = memory ops performed	Decoder causes shadow load/store	Two bit signature
Legend: IFU = fetch unit, IDU = decode unit, TLB = translation lookaside buffer, LSU = load/store unit, I-Cache = instruction cache					

bound demonstrates our design's scalability. This linear scaling with the IPC and the pipeline depth is optimal (up to constant factors) given that we want to monitor every pipeline stage and every instruction.

We experimentally measured how much monitoring network traffic is generated by real programs with two questions in mind: (1) Are there programs that create floods of traffic (near the worst-case bound)? (2) Do high-level differences between programs affect the amount of traffic caused by our monitors? Our expectation was that the different programs would have little impact on the amount of traffic produced by the monitors. As Figure 7 shows, the differences between programs do not significantly impact the EPC (events per cycle) of our system. Figure 7 displays the number of communications per cycle sent between TRUSTNET monitors during executions of SPEC integer benchmarks. These numbers are deterministic because the monitors behave deterministically and the instructions are in order. The traffic generated is relatively low (always less than 2 per cycle). It is also stable across the benchmarks (between 1.1 and 1.2). This supports our belief that a single model works for all programs and that program adaptive features would be unnecessary. These numbers would be higher for a program that, for example, consisted of only store instructions or only branch instructions, but we do not anticipate such behavior in real programs.

F. Area Estimates

In this section, we provide bounds on the general area cost of TRUSTNET and DATAWATCH and estimate the cost of the implementation in our case study. We use bytes of storage as our metric because the computational logic required is trivial (XORs, buffer logic, or equality check over a few bits).

The area cost of our monitors comes from the fact that an event must be stored by the monitoring system from the time it reaches the predictor to the time it reaches the reactor. In complex processors, this time can be variable. It is necessary to have buffers large enough to store all events that are still incomplete. This number depends on the architecture but is

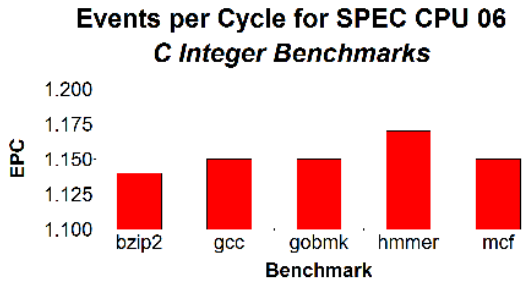

Fig. 7. Events per cycle created by the TRUSTNET monitoring scheme for SPEC benchmarks. An event is any communication between two on-chip units. A prediction and a reaction count as two separate events.

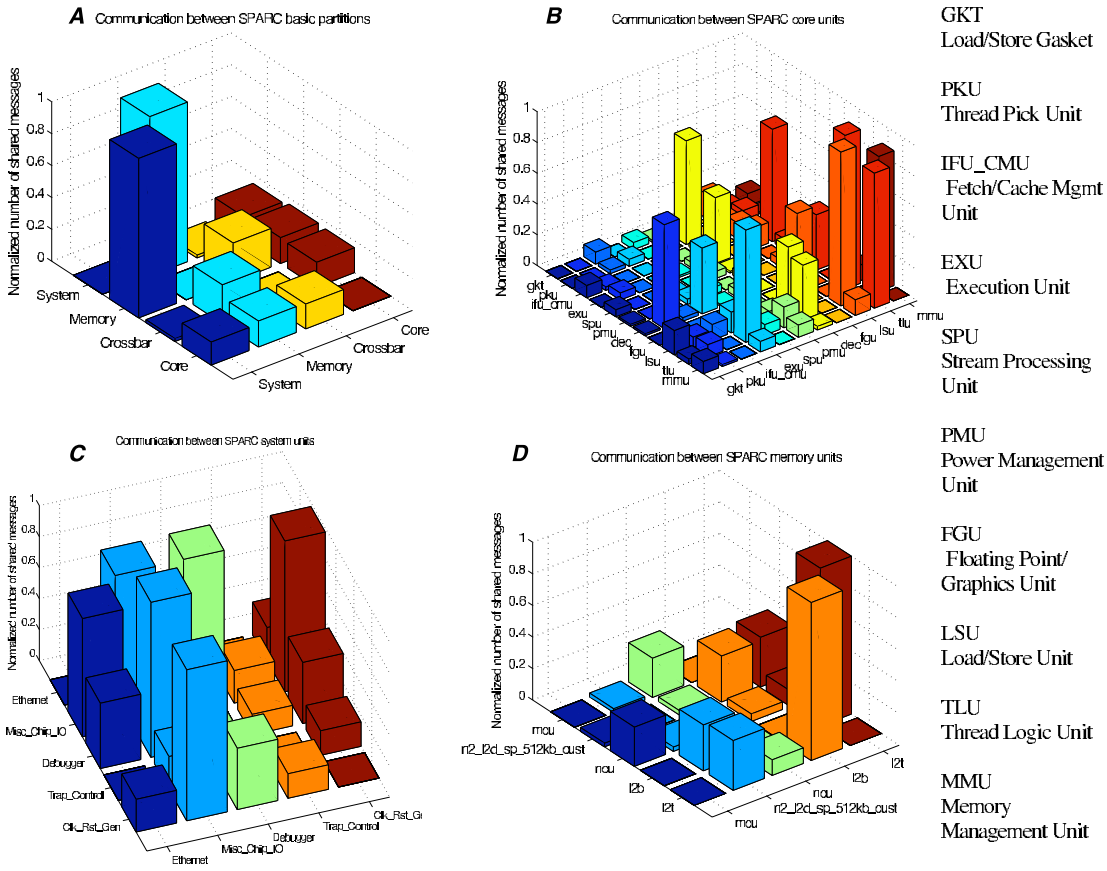

Fig. 6. An overview of the communications that occur in a real OpenSPARC T2 microprocessor. (A) displays a partition of the microprocessor into four basic parts: 'System' includes interfaces, clock generators, and other system level features. 'Memory' cache banks, non-cacheable units, and other memory structures. The core represents one processor core (there are eight cores in all). The crossbar coordinates communications between the cores and the cache banks (which are partitioned on chip). (B), (C), and (D) show internal communications going on within the system, memory, and cores.

known *a priori* for a given microprocessor. Therefore:

$$\text{BufferPackets} \leq \text{MaxMemoryRequests} + \text{MaxInstructionsInPipeline}$$

In the single-issue, in order case, each packet is a single bit. Additionally, if there are N threads sharing a pipeline, the data must be N bits wide instead of one, so that no thread-swapping attacks are possible. So in general:

$$\text{Area} \leq (\text{MaxMemoryRequests} + \text{MaxInstructionsInPipeline}) * \text{PacketSize}$$

Specifically, **TRUSTNET** as described in Table III, employs nine different triangles. It is sufficient to use a one byte prediction buffer for each triangle at the input (although in most cases less would suffice). Analysis of an OpenSPARC T2 core shows that it is impossible for a one byte prediction buffer (eight slots) to overflow. This makes a total of at most nine bytes of storage. Using maximal scaling *i.e.,* conservative scaling with no microarchitectural optimizations, would require $9*8 = 72$ bytes to cover an eight-threaded OpenSPARC T2 core. An OpenSPARC T2 chip, which contains eight cores, would require eight copies of **TRUSTNET** for a total of $72 * 8 = 576$ bytes of storage.

DATAWATCH, as described in Table IV, employs four additional triangles on top of **TRUSTNET**. The two triangles for the pipeline use eight-wide prediction buffers of one byte signatures, for a total of eight bytes each. If we create the two triangles on all eight cores, that makes $2 * 8 * 8 = 128$ total bytes of storage. Including the duplicate direct-mapped TLBs (both data and instruction) adds a total of $128 + 64 = 192$ duplicated TLB entries. If we do this for each of the eight cores and give each line a generous 9 bytes of storage, this adds $8 * 9 * 192 = 13824$ bytes of storage. Then **DATAWATCH** uses a total of $128 + 13824 = 13952$ bytes of storage on top of **TRUSTNET**, for a total of $13952 + 576 = 14528$ bytes, or a little under 15 KB of storage (total for 8 cores and the cache hierarchy).

VI. CONCLUSION

One of the long-standing classic problems in systems security is "How to build trustworthy *systems* from untrustworthy components?" In this paper we study and propose a solution for a variant of the problem: "How to build trustworthy *microprocessors* from untrustworthy components built by untrusted designers?" Since all software and hardware is under the control of microprocessors, establishing trust in microprocessors is a critical requirement for establishing trust

TABLE V

SOME HYPOTHETICAL ATTACKS ON AN INORDER MICROARCHITECTURE. THESE ATTACKS WERE CONCEIVED BY MANUAL ANALYSIS OF THE OPENSPARC T2 RTL (INSPIRED BY [36]) AND IMPLEMENTED IN A SIMULATOR TO TEST OUR DESIGNS. THIS ARRAY OF ATTACKS THREATENS EVERY PIPELINE STAGE AS WELL AS THE MEMORY SYSTEM. THESE ATTACKS CAN VIOLATE CONFIDENTIALITY, INTEGRITY, AND AVAILABILITY. ONLY THE EMITTER AND CONTROL CORRUPTER ATTACKS WERE IMPLEMENTED IN OUR CASE STUDY. THE DATA CORRUPTER ATTACKS ARE DISCUSSED IN THIS PAPER AND PROVIDED HERE FOR REFERENCE BUT WERE NOT IMPLEMENTED.

OpenSPARC Unit	Attack	Possible User Level Effect	Backdoor Type	Protection
IFU	Fetch instruction from wrong address	Fetch a malicious program instead of the one the OS intends.	Control Corrupter	#10
IFU	Fetch extra instructions	Fetch a malicious program in addition to the one the OS intends	Emitter	#2
IDU	Emit spurious instructions	Emit a spurious load or store to private information	Emitter	#1
IDU	Transform no-op into load or store	Allow inappropriate load or store	Control Corrupter	#13
ITLB	Translate pages incorrectly	Translate a valid load into a load from a malicious program	Control Corrupter	#12
ITLB	Change or Ignore permissions	Allow loading from pages without permissions	Control Corrupter	#12
IL1	Loads wrong instruction	Fetch a malicious program instead of the one the OS intends	Data Corrupter	duplic
IL1	Loads extra instruction	Fetch a malicious program in addition to the one the OS intends	Emitter	#4
EXU	Incorrect operation	ALU produces incorrect output; Widespread damage	Data Corrupter	verif. V-C
EXU	Incorrect operation	Compute wrong address	Data Corrupter	verif. V-C
LSU	Loads/Stores extra data	Load/store private information	Emitter	#3
DL1	Loads extra data	Load private information	Emitter	#5#8
DL1	Loads from wrong location in UL2	Load private information	Data Corrupter	duplic.
DL1	Stores extra data	Exfiltrate private information	Emitter	#5#8
UL2	Loads extra data	Load private information	Emitter	#6#7#9
UL2	Loads from wrong location in RAM	Load private information	Data Corrupter	duplic.
UL2	Loads/Stores extra data	Overwrite OS critical information	Emitter	#6#7#9
MC	Loads/Stores extra data	Overwrite OS critical information	Emitter	IV-A
DTLB	Translates data location incorrectly	Translate a valid load into a load of private information	Control Corrupter	#11
DTLB	Change permissions	Allow loading from pages without permissions	Control Corrupter	#11
DTLB	Ignores permissions	Allow loading from pages without permissions	Control Corrupter	#11

in computing bases.

We classified the set of possible RTL level design attacks into three categories and explained the trade-offs between each of the categories. We proposed as a solution to the untrusted microprocessor designer problem **TRUSTNET**, a dynamic verification engine that continuously monitors communications to detect violations of deterministic communication invariants between on-chip units. **TRUSTNET** keeps track of microarchitectural events required to execute an instruction and reports a discrepancy when a microarchitectural unit does more or less work than is expected. We also propose a more robust system, **DATAWATCH**, which watches not only the amount of events that happen but also the type of events that happen. Within these two systems, each unit within a processor is monitored by two other units, a predictor unit and reactor unit. The predictor unit supplies inputs to the actor unit and reactor unit receives outputs from the actor. By tracking predictions and reactions, **TRUSTNET** and **DATAWATCH** detect malicious modifications to a chip.

TRUSTNET and **DATAWATCH** are capable of detecting major categories of microprocessor attacks without complete replication (a classic textbook solution for such problems) at low design complexity, for a small area investment, and with no performance impact. Based on our evaluation of the OpenSPARC T2 RTL, we determined that **TRUSTNET** takes up less than 1 KB of storage to catch emitter attacks. We also determined that **DATAWATCH** can protect the cores and the cache hierarchy from known emitter and control corrupter attacks at the cost of less than 2 KB of storage per processor core. Lastly, we discussed how logic in the rest of the design can be duplicated in order to provide more robust coverage for high security domains at a fraction of the cost of complete duplication (the current state of practice).

The ideas behind **TRUSTNET** *viz.* using the causal structure of microarchitectural operations in concert with the division of work between processor units, opens up exciting opportunities to optimize over traditional techniques used to improve reliability and availability of microprocessors. For instance, **TRUSTNET** and **DATAWATCH** like infrastructure may be used to detect transient faults and for dynamic verification without traditional duplication or diversity based techniques.

VII. ACKNOWLEDGEMENTS

We thank Edward Suh and anonymous reviewers for their detailed comments. We also thank Sal Stolfo and members of the security, architecture and computer systems group at Columbia University for valuable feedback on this work. This work was supported by an instrumentation grant from AFOSR. (FA 99500910389)

REFERENCES

[1] Intel's Silverthorne Unveiled: Detailing Baby Centrino. http://www.anandtech.com/showdoc.aspx?i=3230&p=4.

[2] International Technology Roadmap for Semiconductors 2007 Edition: Design.

[3] Latest from DAC: ST and Media Tek manage media SoC designs (part 2). http://www.edn.com/blog/1690000169/post/290028029.html.

[4] Trusted Computing Group. Online at https://www.trustedcomputinggroup.org/, 2007.

[5] O. Aciicmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *Proceedings of the* 1^{st} *Computer Security Architecture Workshop (CSAW)*, pages 11–18, November 2007.

[6] O. Aciicmez, S. Gueron, and J. P. Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. Cryptology ePrint Archive, Report 2007/039, February 2007.

[7] O. Aciicmez, C. K. Koc, and J. P. Sefert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 312–320, March 2007.

[8] O. Aciicmez, C. K. Koc, and J. P. Seifert. Predicting Secret Keys via Branch Prediction. In *Proceedings of the RSA Conference — Cryptographers Track (CT-RSA)*, pages 225–242, March 2007.

[9] O. Aciicmez, W. Schindler, and C. K. Koc. Cache Based Remote Timing Attack on the AES. In *Proceedings of the RSA Conference — Cryptographers Track (CT-RSA)*, pages 271–286, March 2007.

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

[11] G. Agosta, L. Breveglieri, I. Koren, G. Pelosi, and M. Sykora. Countermeasures Against Branch Target Buffer Attacks. In *Proceedings of the* 4^{th} *Workshop on Fault Diagnosis and Tolerance in Cryprography (FDTC)*, 2007.

[12] 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.

[13] F. Altschuler and B. Zoppis. Embedded system security. January 2008.

[14] D. P. Appenzeller. Formal verification of a powerpc microprocessor. In *ICCD '95: Proceedings of the 1995 International Conference on Computer Design*, page 79, Washington, DC, USA, 1995. IEEE Computer Society.

[15] D. Asonov and R. Agrawal. Keyboard Acoustic Emanations. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 3–11, May 2004.

[16] 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.

[17] 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.

[18] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 259–266, Oct. 2006.

[19] D. J. Bernstein. Cache-timing Attacks on AES, 2005.

[20] E. Biham, Y. Carmeli, and A. Shamir. Bug attacks. In *CRYPTO*, pages 221–240, 2008.

[21] J. Bonneau and I. Mironov. Cache-Collision Timing Attacks against AES. In *Proceedings of the* 8^{th} *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 201–215, 2006.

[22] E. Brickell, G. Graunke, M. Neve, and J. P. Seifert. Software Mitigations to Hedge AES Against Cache-based software side channel vulnerabilities. IACR ePrint Archive, Report 2006/052, February 2006.

[23] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González. End-to-end register data-flow continuous self-test. *SIGARCH Comput. Archit. News*, 37(3):105–115, 2009.

[24] 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.

[25] S. Chatterjee, C. Weaver, and T. Austin. Efficient checker processor design. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 87–97, New York, NY, USA, 2000. ACM.

[26] R. P. Colwell. *The Pentium Chronicles: The People, Passion, and Politics Behind Intel's Landmark Chips (Software Engineering "Best Practices")*. Wiley-IEEE Computer Society Pr, 2005.

[27] J. Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In C. K. Koc and C. Paar, editors, *Proceedings of the* 1^{st} *Cryptographic Hardware and Embedded Systems*, pages 292–302, August 1999.

[28] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, Oct 2001.

[29] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. pages 1–22, 2009.

[30] F. Ferrandi, F. Fummi, G. Pravadelli, and D. Sciuto. Identification of design errors through functional testing. *Reliability, IEEE Transactions on*, 52(4):400–412, Dec. 2003.

[31] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In *Proceedings of* 3^{rd} *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261, 2001.

[32] 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.

[33] T. Harada, H. Sasaki, and Y. Kami. Investigation on radiated emission characteristics of multilayer printed circuits boards. *IEICE Transactions on Communications*, E80-B(11):1645–1651, 1997.

[34] Y. Huang, R. Guo, W.-T. Cheng, and J. C.-M. Li. Survey of scan chain diagnosis. *IEEE Design and Test of Computers*, 25(3):240–248, 2008.

[35] IBM. IBM 4764 PCI-X Cryptographic Coprocessor.

[36] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the* 1^{st} *USENIX Workshop on Large-scale Exploits and Emergent Threats*, April 2008.

[37] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[38] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 9–20, May 1999.

[39] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10), 1973.

[40] 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.

[41] 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.

[42] S. Mangard. Exploiting radiated emissions - EM attacks on cryptographic ICs. In *Proceedings of AustroChip*, 2003.

[43] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Springer-Verlag, Secaucus, NJ, USA, 2007.

[44] V. Marchetti and J. Marks. *The CIA and the Cult of Intelligence*. Knopf, 1974.

[45] J. Markoff. Old Trick Threatens the Newest Weapons. http://www.nytimes.com/2009/10/27/science/27trojan.html?_r=1/.

[46] G. McFarland. *Microprocessor Design*. McGraw-Hill, Inc., New York, NY, USA, 2006.

[47] E. D. Mulder, P. Buysschaert, S. B. Ors, P. Delmotte, B. Preneel, G. Vandenbosch, and I. Verbauwhede. Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem. In *Proceedings of EUROCON*, November 2005.

[48] M. Neve, J. P. Sefert, and Z. Wang. A Refined Look at Bernstein's AES Side-channel Analysis. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, page 369, March 2006.

[49] M. Neve and J. P. Seifert. Advances on Access-driven Cache Attacks on AES. In *Proceedings of Selected Areas of Cryptography (SAC)*, 2006.

[50] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005.

[51] D. A. Osvik, A. Shamir, and E. Tromer. Other People's Cache: Hyper Attacks on HyperThreaded Processors. Presentation available at http://www.wisdom.weizmann.il/~tromer/.

[52] C. Percival. Cache Missing for Fun and Profit. http://www.daemonology.net/papers/htt.pdf.

[53] J. J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *Proceedings of the International Conference on Smart Cards: Smart Card Programming and Security (E-smart)*, pages 200–210, 2001.

[54] 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.

[55] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, New York, NY, USA, 2000. ACM.

[56] K. Rosenfeld and R. Karri. Attacks and defenses for jtag. *Design & Test of Computers, IEEE*, 27(1):36–47, Jan.-Feb. 2010.

[57] 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.

[58] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the Energy Behavior of DES Encryption. In *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, 2003.

[59] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 26–37, Washington, DC, USA, 2006. IEEE Computer Society.

[60] A. Shamir and E. Tromer. Acoustic cryptanalysis: On nosy people and noisy machines. http://people.csail.mit.edu/tromer/acoustic/.

[61] S. Smith. Magic boxes and boots: Security in hardware. *IEEE Computer*, 37(10):106–109, 2004.

[62] 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.

[63] K. Tiri, O. Aciicmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. In *Proceedings of the Fast Software Encryption Workshop (FSE)*, March 2007.

[64] K. Tiri and I. Verbauwhede. A VLSI Design Flow for Secure Side-Channel Attack Resistant ICs. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 58–63, March 2005.

[65] K. Tiri and I. Verbauwhede. Design Method for Constant Power Consumption of Differential Logic Circuits. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pages 628–633, March 2005.

[66] K. Tiri and I. Verbauwhede. A Digital Design Flow for Secure Integrated Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(7):1197–1208, July 2006.

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

[68] S. Vasudevan, V. Viswanath, J. A. Abraham, and J. Tu. Sequential equivalence checking between system level and rtl descriptions. *Design Automation for Embedded Systems*, 12(4):377–396, 2008.

[69] I. Verbauwhede, K. Tiri, D. Hwang, A. Hodjat, and P. Schaumont. Circuits and Design Techniques for Secure ICs Resistant to Side-Channel Attacks. In *Proceedings of the International Conference on IC Design & Technology (ICICDT)*, pages 1–4, May 2006.

[70] 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.

[71] J. Yoo and M. Franklin. Hierarchical verification for increasing performance in reliable processors. *J. Electron. Test.*, 24(1-3):117–128, 2008.

[72] M.-D. M. Yu and S. Devadas. Secure and robust error correction for physical unclonable functions. *Design & Test of Computers, IEEE*, 27(1):48–65, Jan.-Feb. 2010.