Security Analysis and Superscalar Expansion of a Tamper Evident Microprocessor

Robbie Hott, Aleksander Morgan, Benjamin Rodes

December 12, 2010

1 Introduction

Secure software systems ultimately rely on the assumption that microprocessors are trustworthy. By modifying lines of Verilog code, a malicious designer could inject a hardware backdoor, subverting all security software and compromising confidentiality, integrity, or availability of a system. In security and safety critical systems, such as military and airline systems, this is an unacceptable assumption, especially in light of high-profile incidents attributed to microprocessor trojans and backdoor kill switches [1,5].

Given the complexity of processors, the collaborative environment in which they are designed, and the large input space of components, establishing the trustworthiness of a microprocessor is practically infeasible [8]. While there are schemes that attempt to discover possible backdoors prior to system release [3], no guarantees can be made that all malicious hardware has been detected or removed, which makes it advisable to incorporate in the hardware a mechanism to detect possible malicious activity and raise an alarm dynamically.

In this paper we analyze and extend the tamper evident microprocessor design proposed by Waksman and Sethumadhavan [8], in which they attempt to provide trust in a processor (i.e., trust in the processor's behaviour) without a full duplication of components. We scrutinize the level of security the scheme provides, bringing to light the major flaws which we believe provide too many risks for security and safety-critical systems. Despite these issues, we also provide an expansion from the scalar architecture used in their design to a superscalar processor, with emphasis placed on its additional complexity. We believe that while a paranoid approach is best for security, in which no ignored flaws are acceptable, some systems may be willing to sacrifice total security for partial security and efficiency of resources (such as space, performance, and cost).

The remainder of this paper is organized as follows: Section 2 discusses similar works to the tamper evident microprocessor, with section 3 providing an overview of the Waksman and Sethumadhavan's design. Section 4 investigates the security their design provides. Section 5 addresses new assumptions imposed on monitors. Sections 6 and 7 describe our superscalar adaptations. Finally, in Section 8 we provide our conclusions and possible future work.

2 Related Work

The BlueChip system created by Hicks, et al. [3], attempts to extricate possible hardware backdoors by removing unused circuits from the design. Tests from the design verification phase are used to define correct hardware. Any hardware that remains unused during testing is removed from the design and emulated in the BlueChip software. This strategy suffers from false positives and false negatives; it does not guarantee that all necessary hardware remains, nor that all backdoors have been removed.

Another design, proposed by Chatterjee, Weaver, and Austin [2], suggests adding a secondary simple processor that will replicate each pipeline stage of the core processor in parallel. This design validates the final outcome of the untrusted core processor, but does not account for extra instructions or bogus middle stages that could be performed by the core processor.

Waksman and Sethumadhavan [8] propose implementing a run-time test as in [2], but moving most of the testing onto the core processor. They design a tamper evident microprocessor for a simple in-order scalar pipeline, proposing two mechanisms, DataWatch and TrustNet, to be used in conjunction with the simple checker processor in [2].

3 Tamper Evident Microprocessor

The tamper evident microprocessor design of Waksman and Sethumadhavan [8] relies on a multitude of simple monitors to detect malicious activity in both pipeline stages and the memory hierarchy of a scalar processor (resembling OpenSPARC). These monitors are divided into two categories called DataWatch and TrustNet, both of which aim to create a trustworthy CPU containing at most one untrustworthy unit, without fully duplicating the processor (the traditional approach for trustworthy processors [8]).

DataWatch and TrustNet monitor individual units by comparing the predicted output of each monitored unit with its actual output using simple comparison hardware, such as an XOR gate. The unit that provides a monitor with predicted output is called a predictor. The processor unit that receives the actual output from the monitored unit is used as the reactor. A predictor is typically an earlier pipeline stage or lower level cache than the monitored unit; a reactor is typically a subsequent pipeline stage or higher level cache. The monitor triangle structure, as displayed in [8], is depicted in Figure 1. If a monitor detects a discrepancy between the predictor and reactor inputs, an alarm is raised.

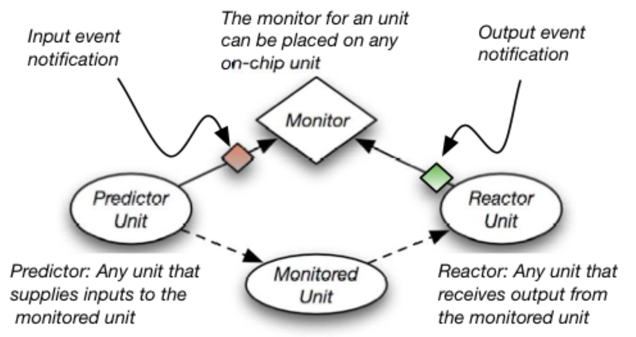
Input event notification
The monitor for an unit can be placed on any on-chip unit
Output event notification
Monitor
Predictor Unit
Reactor Unit
Monitored Unit
Predictor: Any unit that supplies inputs to the monitored unit
Reactor: Any unit that receives output from the monitored unit

Figure 1: TrustNet/DataWatch monitor triangle structure, as proposed in [8].

The authors describe a taxonomy for categorizing hardware backdoors. Backdoors that alter the number of instructions exiting a pipeline stage are referred to as emitter backdoors, while backdoors that corrupt data in any way are referred to as data corruptor backdoors. TrustNet monitors detect emitter attacks; DataWatch monitors detect data corrupter attacks. Other attacks, such as those that rely on environmental factors (i.e., hardware side channels) have previously been investigated, so the authors chose not to handle these for their design.

The TrustNet and DataWatch design is proposed under several assumptions, summarized here:

- An adversary is able to inject a hardware backdoor in only one architectural unit and not across pipeline stages. In other words, cooperating units are not simultaneously lying. This assumption is made given the highly divided and specialized teams of designers. Although these teams do collaborate and could work in tandem for malicious intent, the authors deem it more likely that only a few members of one team are corrupt.

- Backdoors are injected early in the architecture design process, such as during the design specification or the RTL. The authors also assume these injected backdoors are not caught through design reviews.

- Backdoors may not always be active; they may be triggered either by some special input or by a time delay. This further assures the backdoor is not caught through design reviews.

- ROMs written during the microprocessor design phase are statically verifiable.

- The TrustNet and DataWatch monitors are statically verifiable. The monitors consist of a few logic gates and a buffer for queuing up predictor values. In order to build trust, we have to have a foundation of trust on which to build. If we cannot trust the monitors, whose purpose is to generate trust, then the entire monitoring scheme crumbles.

In addition to TrustNet and DataWatch monitors, some duplication is still required. Full duplication of a unit is unavoidable in some situations such as duplication of the ALUs. In other situations, such as for the TLB monitors, the authors propose a partial duplication, or “smart” duplication. Smart duplication avoids the overhead of complete duplication, while still providing the ability to verify a processor unit.

A total of 13 monitors were proposed for detecting instruction tampering in the scalar processor. Of these, 9 were TrustNet monitors and the remaining 4 were DataWatch monitors. The general specifications of these monitors are listed in Table 1 and Table 2. In addition to TrustNet and DataWatch monitors, some duplication of processor units (either full duplication or smart duplication) was used. The authors suggested 21 attacks for their design consisting of both data corrupter and emitter attacks. Of the specified data corrupters attacks, only those which affect control flow, called control corrupters, were actually implemented. The others require duplication to be caught and were therefore omitted since these are outside the scope of testing DataWatch and TrustNet.

4 Security Analysis

For all implemented attacks, TrustNet and DataWatch were successful in attack detection. These attacks were specifically built for the authors design, so these results are not entirely surprising. We now analyze the proposed design and describe the issues and holes that would result in possible undetected hardware attacks.

4.1 Attacks on Availability

Merely recognizing attacks (i.e., detecting malicious tampering) does not provide trust in a system. Since deployed systems cannot be easily or quickly corrected, requiring hardware replacement, a mechanism for handling monitor alarms is necessary, yet absent from the authors' design. They mention using a rollback procedure to resume execution on the last non-malicious instruction, or recording and discarding offending instructions; however, both solutions may stall the pipeline indefinitely. This reduces any detected attack to an attack on availability. The authors note this, but seem accepting of availability attacks over confidentiality or integrity attacks.

Integrity attacks alter the original form of data which can be used for personal gain or sheerly destructive activities. For example, an integrity attack could change a stock transaction from a BUY to a SELL, or vice versa. Integrity attacks may also be used to affect other security properties of systems, namely confidentiality or availability. Because the current design reduces all detected attacks to availability attacks, this means integrity attacks used as a mechanism to affect system

Monitored Unit	Predictor	Reactor	Invariant
IDU	IFU	EXU	# instructions in = # instructions out
IFU	I-Cache	IDU	# instructions in = # instructions out
LSU	D0	D-Cache	# Mem ops issued = # Mem ops performed
I-Cache	F0	L2 Cache	# requested L2 instructions = # of F0 requests
L2 Cache	I-Cache	MMU	# requested instructions = # I-Cache misses
D-Cache	LSU	L2 Cache	# requested L2 data = # LSU misses
L2 Cache	D-Cache	MMU	# requested data from memory = # D-Cache misses in L2
D-Cache	LSU	L2 Cache	# L2 cache lines written = # LSU writes issued
L2 Cache	D-Cache	MMU	# memory lines written = # D-cache writes issued

Table 1: Specifications for TrustNet scalar monitors from [8].

Monitored Unit	Predictor	Reactor	Invariant
IFU	IDU	I-Cache	PC received = PC computed
D-TLB	Checker D-TLB	LSU	TLB output = checker TLB output
I-TLB	Checker I-TLB	F0	TLB output = checker TLB output
IDU	IFU	LSU	memory ops issued = memory ops performed

Table 2: Specifications for DataWatch scalar monitors from [8].

availability will always succeed. Consider a missile guidance system, or a nuclear power plant's core temperature monitoring system. If the attacker's intent is to alter computed data such that a missile misses its target, or a nuclear power plant melts down, he could do so either by generating incorrect data in hardware (an integrity attack aimed toward availability), or by making the hardware unavailable (a strict availability attack). This makes integrity attacks of this kind particularly problematic.

Presumably, an attacker would want any hardware backdoors to be sophisticated enough to allow for full access to a system and/or recovery of sensitive information. These attacks typically attempt to be covert. An attacker, however, may wish only to affect only system availability, which is highly overt. A backdoor designed for this purpose alone can simply fail after a given time period. Since the current design reduces all attacks to availability attacks, an attacker could conceivably mitigate the overt nature of these attacks by obfuscating the location of the attack if the malicious unit is a predictor or reactor for some other unit. That is, if an attacker can manipulate data sent to a monitor, then another unit will be flagged as malicious, and an availability attack will still occur. This type of attack may not be possible, depending on the exact nature of the design, and what assumptions can be made concerning the static verifiability of how predictor or reactor units send data to monitors.

To prevent the reducibility of every attack to an attack on availability, which consequently prevents availability attacks in general, some kind of recovery mechanism is required. Recovery can be accomplished through duplication of the offending unit so that execution may continue with correct data. Typically, at least three processors are required to determine exactly what data should be used for recovery. If TrustNet and DataWatch could be trusted to catch all attacks, then this monitoring mechanism can be used in conjunction with duplication to single out the correct processor unit without the need for more than two processors. However, the scope of TrustNet and DataWatch is limited to, for the most part, the pre-execution stages of the pipeline. The units that are only monitored by duplication, such as the ALUs, will require at least three duplicates, as there is no mechanism to determine which was correct under the current design.

Ultimately, malicious availability attacks are observably no different than non-malicious hardware failure, and must be handled in the same fashion: hardware duplication. Since availability specific attacks could target any portion of any particular unit, including the entire unit, only full duplication of every processor unit will prevent these attacks.

4.2 Attacks on DataWatch

DataWatch appears to be rather problematic in general. Not only are DataWatch monitors missing for most units covered by TrustNet, the units that are monitored have theoretical attack loopholes, due to both the lack of monitor coverage and the nature in which monitoring is performed.

The emphasis of the author's design appears to be on TrustNet. There are only 4 suggested DataWatch monitors, leaving significant gaps in the processor security. TrustNet protects the IFU, IDU, LSU, as well as various elements of the memory hierarchy, yet the LSU and all of the memory hierarchy are not monitored by DataWatch. It is most shocking that the LSU is not protected in some way, since this unit interacts with process state, and any unit not covered could conceivably manipulate both instructions and data in any fashion. Even if TrustNet can be fully trusted, it will be rendered practically useless if these unmonitored units can change an instruction or parameter arbitrarily.

For those units that are covered by DataWatch, the way in which the unit is monitored does not cover all angles of attack. Any processor unit which provides unique manipulation, interpretation or execution of instructions will require duplication to guarantee complete trustworthiness. Consider the IDU. The IDU alone is responsible for interpreting and fetching parameters of instructions. A malicious IDU could insert or alter parameters for an instruction which no other pipeline stage could detect without duplication. The authors attempt to catch alterations to the instruction itself, such as changing an ALU instruction to a memory instruction, but alteration to the instruction's parameters are unaccounted for and cannot be caught without duplication.

The IFU suffers from a similar problem. DataWatch only verifies the PC of the instruction received by the IDU, not the actual instructions that are passed to the IDU. A malicious IFU could falsify an instruction while making the PC appear to be genuine (i.e., the expected PC). The IDU would receive the proper PC and an invalid instruction, raising no alarms.

The way in which DataWatch captures values to be monitored is also potentially problematic. DataWatch takes a signature of a monitored value for comparison by a monitor instead of the complete value. Without monitoring the entire value in question, a malicious unit may be able to alter a value such that the signature is still the same, thereby eluding detection. For example, in the suggested design, a two bit signature is used to protect against the IDU from changing an instruction. A malicious IDU could change the instruction such that the signature is unaffected. The manipulated data will at least cause an availability attack if the manipulated value results in a runtime exception (i.e., the resulting malicious instruction is not valid). However, it is conceivable that a signature collision could exist that produces valid and useful instructions for an adversary. For instance, given a sequence of malicious instructions that must be executed to implement an attack, say with signatures 11, 10, 01, and 11. The IDU can compute the signature of each incoming instruction, and replace the first instruction having signature 11 with the first malicious instruction, replace the next instruction having signature 10 with the next malicious instruction, and so on.

Cryptographic hashes of entire instructions or operands, in which collisions are highly improbable, are likely not a viable option because of both the small size of the monitored values (perhaps only a few bytes) and the latency a hash of this kind will impose on the processor. A better approach would be to use the entire value for monitoring, which may have overhead issues as well; however, this is probably the simplest solution.

4.3 Attacks on TrustNet

The concept of TrustNet seems sound to provide adequate protection under its purpose of detecting emitter attacks. It appears to be difficult to inject or delete instructions from the IDU or LSU without detection; however, the same cannot be said for the IFU.

It is not clear how the TrustNet monitors for the IFU and memory hierarchy work. The IFU monitor description states that it assures that the unit only sends to the IDU as many instructions as were fetched from the instruction cache. Perhaps a malicious IFU can simply not fetch from the instruction cache, thereby deleting instructions from the pipeline. Alternatively, the IFU could fetch more instructions from cache, but then ignore them to add bogus instructions while correcting the PC, as described in the section on DataWatch attacks, to prevent the IFU DataWatch monitor from triggering an alarm. This kind of attack may also be possible with units in the memory hierarchy. Again, it is not clear how these monitors work, so it is difficult to gauge if this kind of attack is even possible under the current design.

4.4 Register Write Back

At some point in the pipeline, values must be written back to registers. Yet, no mention of register write back is made either by TrustNet, DataWatch, or any duplication mechanism. Depending on the actual implementation, a dedicated write back unit may not even exist; however, some unit must be the last to handle a piece of data and to route that data to a register. Under the current design, there is no reason to assume data will arrive in the register file as intended.

Trying to devise a DataWatch monitor for write back, whether or not it is a dedicated unit of the pipeline, may be problematic since there are no subsequent stages to act as a reactor. A monitor could be fabricated, at the cost of an additional pipeline cycles, which would compare what is stored in a predicted register with what is actually present. This monitor is more complicated than the originally proposed monitors, and therefore may be corruptible if it cannot be statically verified. It may be possible simply to tap the data sent to the register file for comparison against an expected value. However, since the actual register is never checked, it may still be possible to store invalid data or send the data to an invalid location without detection.

A TrustNet monitor may still be conceived for write back, assuming that any signal sent to the register file can be monitored. The TrustNet monitor would then raise an alarm if a unit performs a write back when no request was actually made.

4.5 Analysis Summary

The major struggle in properly analyzing the Waksman and Sethumadhavan design is the lack of real world, well documented hardware backdoors. Some attacks have been attributed to hardware trojans [5], while other attacks have been academically proposed, both simple and sophisticated [4]. Without a study into the reality of hardware backdoors, it is hard to argue the authors' scheme provides adequate security due to a theoretical lack of protection. That is, some of the risks present in the design may, in reality, be infeasible to exploit due to their innate complexity, likely detectable prior to hardware release. There is no data to support this, however. The authors even speculate that data corrupter attacks are unlikely, yet there is no convincing evidence.

Additionally, the authors' design was very difficult to analyze due to lack of detail. The design specifics are described rather abstractly, with the exact nature of their pipeline and memory hierarchy not clearly defined, leaving us in the dark as to how secure the system actually is, and what kind of attacks are still possible.

Based on the design present in the authors' paper, we have shown the tamper evident design leaves open many possibilities for attacks, and therefore may have to be discarded in favor of complete duplication of the processor. It is preferable to avoid duplication, especially multiple duplication as we suggest is required for attack recovery, in order to balance cost of resources and desired security goals. There is a degree of security that is provided in this design, as it is able to catch the attacks used in the authors' validation, therefore there may still be some merit in the design concept. In fact, we believe the concept of TrustNet is fairly sound and could be made to catch all inserted or deleted instructions under the same design assumptions the authors made, even though this mechanism has a somewhat limited attack space coverage.

5 Generalization of Monitors

Accepting the security risks for a less expensive yet somewhat secure processor, we now extend the TrustNet and DataWatch monitor design onto a more viable processor. The initial conversion is to a simple superscalar processor, Sun's SuperSPARC. After expanding to TrustNet and DataWatch to the SuperSPARC, we would also like to add additional monitors to cover the textbook n-wide superscalar processor. Since an n-wide academic processor provides register renaming, a reorder buffer, and out-of-order execution, we must address each of these new functionalities and how they can be secured. These additional functionalities require us to generalize the definition of monitors as proposed by Waksman and Sethumadhavan.

To do so, we first rephrase the idea of a monitor and discuss the assumptions that are implicit in the original design, but need to be made explicit and generalized for our design. The result of this section is a modified monitor description and design assumptions that will be adequate to handle the complexities of superscalar designs.

5.1 Rephrased Monitors

Sethumadvan and Waksman use the concept of a monitor to make sure that all the processor calculations are trustworthy. Their monitor receives some information from one unit (called the predictor), holds that data for a specified number of cycles, receives data from another unit (called the reactor), and performs a calculation to make sure the two pieces of data are consistent. In doing so, it monitors one function of a particular unit, called the monitored unit. The units with multiple functions, such as the L2 data cache, have multiple monitors present, namely read and write monitors.

This notion of a monitor is inadequate for handling the complexities of the superscalar design. While most of the TrustNet monitors tend to generalize easily to superscalar, as described in Section 6, DataWatch monitors become more complicated due to the non-deterministic function of some units which make decisions based on heuristics. As a result, in order to generalize the notion of a monitor to a superscalar design, we extend several definitions.

While the original monitors held on to data in latches for a specified number of cycles, our generalized monitor can hold on to data indefinitely, in data storage structures. This generalization is necessary because the allowing of out-of-order completion makes it impractical to try to predict the latency of an individual instruction through the pipeline.

In addition, the original concept of a monitor received the data only from two sources, a predictor and reactor, one a number of clock cycles after the other. The alarm is sounded if the data from the reactor does not match the data expected based on the input of the predictor. On the other hand, our generalized monitor concept may receive data from more than two different sources. This monitor may receive information not only from multiple predictors and reactors, but also have its data structure updated by other units in the pipeline.

With these generalizations in mind, we discuss the implicit assumptions of Sethumadvan and Waksman.

5.2 Simplicity Assumption

Sethumadvan and Waksman had several assumptions that they did not make explicit. Because our design needs to generalize those assumptions, we must make them explicit. One of the assumptions glossed over in the original design was the simplicity argument. It implies that while the designers are not sure of trustworthiness of all other units, the monitor can be trusted to:

1. receive information properly, since it can be manually verified that the input wires come from the right places,
2. hold on to information properly, which for the original design is assumed because the information is held in simple latches, and
3. perform the comparison correctly, since the logic performed is fairly simple, and its correctness can be checked by hand.

This simplicity argument is particularly important, since if the monitor cannot be trusted, then not only can it allow other attacks to succeed, but also it becomes a key vulnerability, opening the door to many attacks. As a result, before any claim of utility of the generalized monitor is asserted, the simplicity argument will have to apply to it as well. We make the following justifications for this assumption:

1. The generalized monitor can receive information properly for the same reason as the originally proposed monitor.
2. The generalized monitor can hold on to information properly if we make sure that the only write ports to the data storage cells come from the proper source. This is, in essence, a part of the "statically verifiable storage" assumption from the original design.

3. The generalized monitor can perform the comparison correctly because, as with the original monitor design, the comparison is performed via a simple XOR gate or a small collection thereof.

4. In addition to the above three, we need the generalized monitor to be able to find the correct address in the monitor data storage structures correctly. We can assume this can be verified statically as follows:

   (a) we need to make sure that the wires to the monitor run from the correct data storage structure, and

   (b) the pointer needs to go to the right target, so the wires indexing the target unit need to be checked for correct redirecting.

Since we assume these four actions are statically verifiable, our generalized monitor system can be assumed and tested to be free of malicious tampering.

5.3 Safe Speculation

Another design assumption that is implicit in the original design is the safe speculation assumption. In the original design, the predicted output of a given monitored unit arrived at the monitor at the beginning of the clock cycle, with the actual output arriving within one clock cycle after that. Since the monitor is only a simple XOR gate, the result–and consequently, the alarm–is attained very quickly, within a cycle after the instruction passes through the monitored unit. In our design, however, the monitors are more complicated and may require cache lookups and more complex comparators, such as multiple AND, OR, and XOR gates. Therefore, the monitor may take more than one cycle after the output has arrived from the reactor. In that case, we have the options of halting the execution while the monitor checks, or assuming that the instruction is safe, with the monitor's alarm raised a cycle or two after the fact. We assume the latter, calling this assumption safe speculation. Since the alarm is not sounded until the instructions are later in the pipeline, the recovery must be able to figure out which instructions to flush from the pipeline, and how to re-execute them without deviating from the sequential architectural model.

5.4 Redundant Inconsistency

Finally, an assumption that we make, which is unique to superscalar designs, is the assumption of redundant inconsistency. Each of the processor units can be viewed as a black box as long as the actions it performs are consistent with each other. For instance, it is possible that when a unit stores a value in the RRF to register 2, it actually gets stored in register 1, and vice versa. However, as long as the references to the registers are consistent, or in other words as long as the processor receives the value of register 1 every single time register 2 is read, and vice versa, then the sequential architecture model is not violated. We will state on which data field of any unit in which redundant inconsistency is applied, whenever this assumption is invoked.

6 SuperSPARC Adaptation

The SuperSPARC processor [7] is a simple superscalar processor based on the SPARC v8 instruction set, using a four-cycle integer pipeline. The two-phase fetch cycle (F0, F1) fetches up to four instructions per cycle, placing the fetched instructions into an instruction queue. The decode stage (D0, D1, and D2) issues up to 3 instructions per cycle into either the floating point unit or the integer execute stage. The Integer execution stage (E0, E1) performs both data memory accesses and 2-stage ALU operations. The pipeline concludes with the write back stage, in which data is written to the register file and stores are retired into the store buffer. Figure 2 shows a simplified view of the SuperSPARC processor, while Table 3 gives a detailed overview of the pipeline stages. The pipeline also supports both forwarding and cascading of dependent data.

TrustNet and DataWatch can easily be extended to this processor, since it closely follows the model of a scalar processor, with one interesting catch: it fetches up to four instructions per cycle, but decodes only up to three. It does this based on the following groups [7]:

- Maximum two integer operations
- Maximum one data memory reference
- Maximum one floating point instruction
- A group is terminated after each control transfer.

6.1 Simple Extended Monitors

Since the SuperSPARC processor closely resembles a scalar processor with respect to the memory hierarchy and TLB lookup strategies, we can easily extend the monitors from the original Tamper Evident Microprocessor to the SuperSPARC. These simple monitors include all the TrustNet monitors for LSU, I-Cache, D-Cache, and L2 Cache, as well as DataWatch monitors for the Data- and Instruction-TLBs. See Tables 4 and 5 for the full listing of monitors.

6.2 Adapted Monitors

Not all monitors can be easily extended to this new architecture. We must adapt the fetch and decode monitors to handle differing numbers of instructions per cycle, since the original design assumed only one instruction through each pipeline stage (in order). These adaptations can be simple, since the SuperSPARC provides in-order fetch and decode: the fetch and decode stages both happen before execution and cleanup of out-of-order completion. A summary of the updated monitors appears in Table 6.

6.2.1 Instruction Fetch Monitors

The SuperSPARC processor has two instruction queues: one for branch taken, and one for branch not taken [6]. As a result,

Stage	Activities Performed
F0	I-cache RAM and TLB lookup
F1	I-cache match detection and 4 instructions sent to instruction queue
D0	1-3 instructions issued, load/store address registers selected
D1	Read load/store address registers, allocate ALU resources, evaluate branch target address
D2	Read ALU operands, calculate effective addresses
E0	ALU stage 1, Data lookups, FP dispatch
E1	ALU stage 2, D-cace match detection, loads completed, exceptions resolved
WB	Write back to register file and retire stores into buffer

Table 3: SuperSPARC's pipeline stages, as described in [7].

Monitored Unit	Predictor	Reactor	Invariant	Example of attack thwarted
LSU	D0	D-Cache	# Mem ops issued = # Mem ops performed	LSU performs shadow loads
I-Cache	F0	L2 Cache	# requested L2 instructions = # F0 requests	I-Cache returns spurious instruction to IFU while waiting on the L2 Cache
L2 Cache	I-Cache	MMU	# requested instructions = # I-Cache misses	L2 Cache returns suprious instruction while waiting on main memory
D-Cache	LSU	L2 Cache	# requested L2 data = # LSU misses	D-Cache returns false data while waiting on the L2 cache
L2 Cache	D-Cache	MMU	# requested data from memory = # D-Cache misses in L2	L2 Cache returns spurious data while waiting on main memory
D-Cache	LSU	L2 Cache	# L2 cache lines written = # LSU writes issued	D-Cache sends write to L2 cache unprompted
L2 Cache	D-Cache	MMU	# memory lines written = # D-cache writes issued	L2 sends write to memory unprompted

Table 4: TrustNet scalar monitors that are extended to SuperSPARC.

Monitored Unit	Predictor	Reactor	Invariant	Example of attack thwarted
D-TLB	Checker D-TLB	LSU	TLB output = checker TLB output	TLB violates permissions
I-TLB	Checker I-TLB	F0	TLB output = checker TLB output	TLB violates permissions

Table 5: DataWatch scalar monitors that are extended to SuperSPARC.

Monitored Unit(s)	Predictor(s)	Reactor(s)	Invariant	Example of attack thwarted
F0, F1	I-Cache	I-Queue	# instructions in = # instructions out	Fetch loads branch instructions when no branch is present
F0, F1	D1, I-Queue	I-Cache	PC received = PC computed	F0 fetches instructions from a previous branch address
D0	I-Queue	FPU, I-File	# instructions in = # instructions out	Both branch-taken and not-taken IQs send instructions to decode
D0	F1	FPU, I-File	# memory ops issued = # memory ops performed	An ALU op is converted by D0 to a write

Table 6: Updated TrustNet and DataWatch monitors for the SuperSPARC processor.

the monitors handling the instruction fetch and decode will be more complicated than in the original design.

#1 IFU (F0, F1): The TrustNet monitor for the fetch unit, stages F0 and F1, works similar to that of the original authors, except we must use the instruction queue as the reactor, since the decode unit may only decode one instruction during the next cycle. Therefore, when a 4-instruction line is read out of the instruction cache, the count (3 bits) and the target

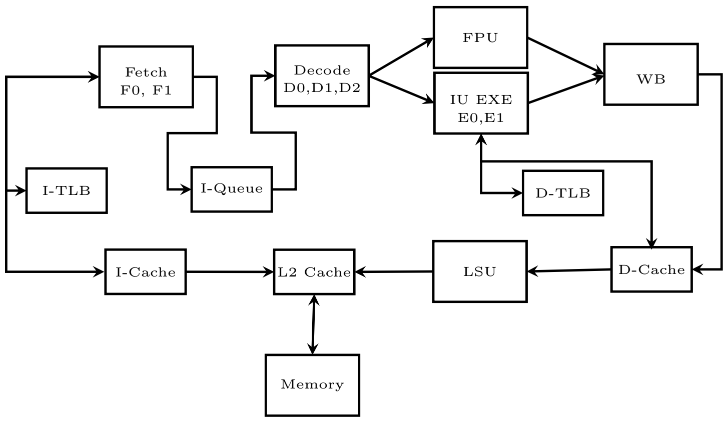

Figure 2: Simplified diagram of the SuperSPARC processor showing unit communication, as considered in this paper.

instruction queue (1 bit) is sent to the monitor. Likewise, the change in size of each I-Queue is read during the next cycle and sent to the monitor. This monitor computes that the change of the non-target I-Queue size is unchanged, and that the change of the target I-Queue size is equal to the count.

This design requires four wires connecting the predictor and the monitor, rather than just one. Similarly, it uses a count from both instruction queues, requiring 8 more wires on the reactor side. But, since these only grow with the width of the pipeline, they should perform similar to the 2 wires in a scalar pipeline. The monitor, since it is more complex, will take more than one XOR gate, but should not require more than a few gates to enact.

#2 IFU (F0, F1): The DataWatch monitor for the fetch unit becomes a bit more complicated, since we currently have branch prediction and multiple instructions in the pipeline at any given time. We still want to ensure that if the instruction cache receives a valid PC, it follows in program order from previous instructions or is a valid branch address. Therefore, the reactor will still be the I-Cache, which receives the final PC value to fetch. The predictor and monitor, on the other hand, must become more complicated. The D1 pipeline stage and the instruction queues will predict the next PC value from the fetch unit. Moreover, some duplication of the PC logic will be necessary in order to compute the next sequential PC. The entries written to the instruction queues will be forwarded to a predictor unit, which will only use the last entry's PC to compute PC + 4 and attain the next sequential PC. That value, and the branch target address computed in the D1 stage will be forwarded to the monitor. The monitor then tests that the I-Cache's received PC equals one of the two actual PCs, either the calculated branch address or the next sequential instruction.

As a note, we delay the actual testing of the PC in the monitor by at least one cycle, since we must wait for the branch target address to be calculated. This is a safe assumption because the branch target address will be computed before any incorrect instructions are decoded, since the instructions are scheduled in order. Due to this assumption, this monitor also serves to verify the branch predictor.

This design only requires one additional PC to be sent to the monitor, in addition to the original predictor and reactor PCs from the initial scalar design, which in turn is required to perform one additional test. Therefore, it should also perform as well as the original.

6.2.2 Instruction Decode Monitors

#3 IDU (D0): The TrustNet monitor for the decode unit, stage D0, also performs similar to that of the original authors, except it only monitors the first stage of the decode cycle. For TrustNet, we are only ensuring that the number of instructions does not change, therefore we only need to monitor the first stage of the decode unit which handles the issuing of instructions. The I-Queues predict the D0 stage, and the number of instructions removed from each queue in a given clock cycle is forwarded to the monitor (2 bits per queue). The floating point unit and integer execution stages react to the decode stage, sending the number of instructions issued for each cycle to the monitor. The monitor then checks for two possible scenarios. First, the 2 bits from each instruction queue are compared to ensure that the decode unit did not schedule from both queues, raising an alarm if so. Second, it computes that the OR of the counts of the two queues (the total instructions removed) is equal to the number received from the reactors. Note: unlike the original authors, we cannot use the IF to predict since that unit fetches up to 4 instructions, but the decode stage only schedules 1, 2, or 3.

This design requires four wires connecting the predictors and monitor, rather than just one, since the monitor must receive inputs from both queues. Similarly, it requires a count from both the integer and floating point units, requiring another 4 wires on the reactor side. But, since these only grow with the width of the pipeline, they should perform similar to the 2 wires on a scalar pipeline. The monitor, though, must be able to test for equality using XOR gates, combine inputs using OR gates, and perform these in a specific combination. It, like the fetch monitors, only requires a small amount of additional gates, with nothing very complicated, and therefore should perform as well as the scalar version.

#4 IDU (D0): For the DataWatch monitoring of the decode unit, in order to match the decode DataWatch monitor of the original design, we want to ensure that no instructions were erroneously changed into memory operations by the D0 stage of the decoder. Since we are using the SPARC v8 instruction set, this test simply requires verifying the first two bits of the instruction fetched. Memory operations have these first two op bits set to 11 [6]. Since the processor fetches and decodes multiple instructions in order, we introduce a simple FIFO predictor unit that stores the first two bits, a token, of each instruction fetched. These bits are stored when the F1 unit stores the instructions into the active instruction queue. The floating point unit and integer execution stages react to the decode stage, sending the monitor the type of instruction decoded. With very simple logical gates, the monitor can test

that the first predictor token equals the token from the reactor and validate that the number of memory instructions did not change. Although we only monitor the first two bits of the instruction to match the original author's design, this could be expanded to include more bits, up to the entire instruction.

This monitor, like its scalar version, is one of the more complex monitors discussed so far. It must be able to access the first two bits of each instruction as sent by the predictor. The predictor unit must have four write ports to allow all four instructions' first bits to be inserted into the array in parallel. The monitor may compare in the same fashion as the scalar version, linearly, or it may utilize 3 read ports from the array to XOR up to three instructions per cycle. Note that the latter would require our monitor to be 3 times as wide as the original.

6.3 SuperSPARC Adaptation Summary

SuperSPARC provides an easy adaptation of TrustNet and DataWatch to a superscalar processor. Its design allows for most of the monitors to be expanded directly from the scalar pipeline proposed by Waksman and Sethumadhavan. However, we did have to tweak four monitors to achieve the same attack space coverage as the scalar version. These updated monitors still hold to most of the original design's assumptions and are all still simple. Therefore, these monitors can be statically verifiable and do not significantly increase the processor's complexity. We now must send 4 bits per clock cycle instead of 1 for monitors 1 and 3 and increase the buffer for monitor 4 slightly to handle the additional load due to multiple instructions queueing at one time. Monitor 2's predictor unit, which in the original design duplicated the PC logic, must still compute the next PC to fetch. However, for the SuperSPARC processor, it not only computes the PC for the next line to fetch, but must also keep track of the branch predictor's prediction. Therefore, since the monitor assumptions hold and are on the same order as the scalar versions, this design should perform as well as the original design and cover the same attack space.

7 Generalized Superscalar Adaptation

In order to extend to the textbook n-wide superscalar pipeline, we note that we can build upon the monitors we provided for SuperSPARC, adding an instruction queue to this generalized pipeline, as depicted in Figure 3. If the processor always fetches and decodes n instructions per cycle, we would be able to use a simpler (as compared with the SuperSPARC monitors) n-wide version of Waksman and Sethumadhavan's scalar monitors for both fetch and decode. Since instruction fetch and decode are accounted for, this leaves monitoring register renaming and the reorder buffer, which allows out-of-order execution, to address.

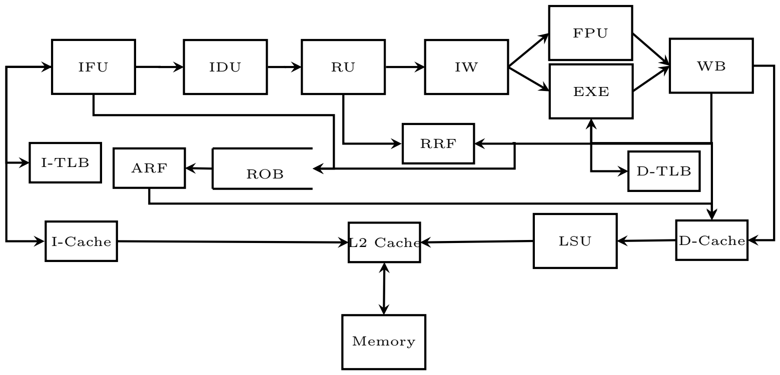

Figure 3: Simplified diagram of the textbook superscalar processor showing unit communication, as considered in this paper.

7.1 Register Rename Monitors

Register renaming creates a security risk when expanding TrustNet and DataWatch, since the authors' scalar processor only uses one register file. In order to simplify the design, we will assume that throughout renaming, our superscalar pipeline fetches, decodes, and renames n instructions per cycle in order. The set of hardware gates which implement assigning an available rename register to each instruction's destination ARF register before that instruction proceeds to the Issue Window (IW) will be called the Rename Unit (RU). This unit is also responsible for ensuring that future instructions dependent on that computed value as a source operand will be given the correct rename register. If we can verify that registers have been renamed correctly without overwriting an in-use register and that the correct source operands are assigned from the RRF, then the pipeline up to the IW resembles the authors' scalar pipeline, and all their assumptions with relation to security apply. We propose the following two monitors to handle this task:

RRF register	Busy bit
t_1	0 or 1
t_2	0 or 1
$\vdots$	$\vdots$
t_n	0 or 1

Table 7: Destination operand reassignment monitor array.

# 5 RU (destination renaming): The first monitor we propose is to guarantee that the rename unit does not rename an instruction's destination ARF register to an RRF register that is currently being used. That is, a rename register cannot be reassigned until after its value has been written to the ARF. This monitor must utilize smart duplication of the RRF, allowing it to validate the entries that are being assigned in the real RRF. The monitor will include a direct-mapped cache, which we refer to as an array, as depicted in Table 7.

Each instruction's destination operand will be forwarded to the monitor as it is written into the IW. The monitor will use the destination RRF register received to index into its array and obtain the busy bit. If the busy bit is 1, the alarm will be thrown; if the busy bit is 0, no alarm will be thrown and the monitor must update the busy bit to 1. Also, the busy bits will be set to 0 during the commit stage, when the instruction is retired from the ROB and the value of the RRF register is written back to the ARF.

Therefore, this monitor is predicted by its own smartly duplicated RRF array. Its reactor is the output of the is the rename stage, which will forward to the monitor the destination RRF register for the instruction on its way to the IW. We can use the monitored unit as its own reactor by the redundant inconsistency assumption, since an instruction will not be renamed again once it leaves the rename unit and monitor 6 ensures that the assignments are consistent. Consequently, since the monitor's alarm is expected at a later pipeline stage, rather than at the rename stage as the original TrustNet design implied, we only catch an erroneous renaming at least one cycle after that renaming has occurred. As a result, the pipeline needs a more complicated recovery process, since recovery from alarms of this monitor would involve flushing instructions that may already be in the execute stage.

ARF register	RRF register assigned
r_1	t_i
r_2	t_j
$\vdots$	$\vdots$
r_m	t_k

Table 8: Source operand renaming monitor array.

# 6 RU (source operands): Not only must we ensure that busy registers are not allocated, but we must also monitor that once a destination ARF register has been allocated an RRF register, that mapped assignment propagates to future reads of the ARF register until the register is reassigned. Therefore, we propose a monitor to ensure that the renaming of source operands happens correctly. This monitor, like the previous, must also utilize smart duplication to ensure the latest assignments of the correct registers are being used. It will include a direct-mapped array of ARF to RRF assignments, indexed by the ARF register number, as seen in Table 8. When a destination register is assigned during the rename phase, the monitor's table is updated to reflect the current mapping. If the mapping is invalid, monitor 5 will detect an incorrect mapping and raise an alarm.

Since we are monitoring the renaming of source operands, the IDU will be used as the predictor. It will send to the monitor the ARF registers it decoded as source operands of the current instructions. As the rename stage is being performed, the monitor will resolve the ARF entries to their current RRF assignments using its array. Since we are assuming in-order decode and rename, the monitor will see the register values for the destination operands before the rename of the source operand is complete. Once the instructions pass through the rename phase, their renamed operands, as sent to the IW, will be forwarded to the monitor and serve as the reactor. This monitor will then compare the predicted RRF entries needed with the RRF entries assigned, raising an alarm on a mismatch.

7.1.1 Reorder Buffer Assumptions

We choose not to monitor the reorder buffer (ROB) due to our simplicity assumption. Since the ROB is a storage unit, we assume it can be statically verifiable. Any unit that would be able to corrupt the ROB would have to do so through standard pipeline channels, such as during updates to reorder out-of-order execution. Since we cover register renaming with monitors 5 and 6, as well as the decode and fetch stages with the inherited monitors from the SuperSPARC design, we feel it is safe to assume that until an instruction is issued into the IW, the ROB has been updated correctly.

7.2 Generalized Superscalar Summary

In order to cover the same attack space for our generic superscalar processor as Waksman and Sethumadhavan, we created two new monitors to cover register renaming to ensure that the instructions are set up correctly before being executed. The monitors, however, required new assumptions to be made on what a monitor is and how complex it is allowed to be. Also, these monitors required vast amounts of duplication, each storing an entry for every register in the RRF or ARF, either indexed by the RRF register ID or the ARF register ID. Assuming this amount of duplication is feasible on a given chip, the 14 monitors we propose protect against the same attacks as the scalar processor since the 12 for SuperSPARC provide that level of protection and the two new monitors ensure that operands of each instruction successfully arrive at the execution stage of the pipeline.

To ensure security beyond the rename stage would require full duplication of the ROB and some execution units, as the scalar version required duplication of the execution stage, making TrustNet and DataWatch as expensive as full duplication for superscalar pipelines. Even providing the same level of security as Waksman and Sethumadhavan, this design required partial duplications of the RRF and ARF arrays and non-trivial monitors: far more duplication than the original design.

8 Conclusions and Future Work

This paper extended the design of a Tamper Evident Microprocessor to a superscalar version. To ensure completeness, we applied the design to a SuperSPARC processor. Most of the TrustNet monitors were simply extended to match the width of the pipeline. Two TrustNet monitors had to be customized to meet the design of SuperSPARC. Also, two of the DataWatch monitors were extended, whereas two others were customized. We then proposed a way to modify this design

Monitored Unit(s)	Predictor(s)	Reactor	Invariant	Example of attack thwarted
RU	the monitor's data storage array	output of RU to IW	RRF registers in use are not reassigned	RU assigns a second ARF register to an RRF register
RU	IDU	output of RU to IW	source operands correctly renamed	RU directs a source operand to use another register's value

Table 9: New TrustNet and DataWatch monitors for the generic superscalar processor.

to an academic superscalar design, which includes monitoring a nondeterministic Rename Unit. In order to ensure the generalization was sound, several assumptions from the original design had to be spelled out and extended.

The designs we proposed suffer from the same problems as the original design. For instance, all attacks can still be reduced to an availability attack, no instruction operands can be monitored without heavy duplication, and nothing starting with the EXE stage is monitored. Even monitoring of the post-EXE processes, such as committing of RRF registers, is limited to verification of correctness during IDU, and depends on trustworthiness of the EXE stage and the ROB. However, our designs do cover no less than the original design. In particular, assuming the trustworthiness of the ROB, no new risks are created by allowing register renaming and full out-of-order execution.

Assuming our superscalar expansion suggestions are sound under our assumptions, and provide a viable solution to some users (i.e., the reduced security compared to full duplication can be accepted), an actual implementation of our design on hardware is left for future work. The logical next steps in the process would be a simulation of these new monitors in a software superscalar simulator, followed by detailed plans for a current processor. However, a more-detailed low-level cost analysis of our proposed design in terms of additional hardware, power consumption, etc. is probably the most prudent initial follow on to this work to determine if our design is even desirable in terms of these properties.

Additionally, further expansions are also left for future work, including the addition of handling side channel attacks, recovery mechanisms, and a design to handle the possible corruption of multiple pipeline stages. It is also worth noting that additional research into the current status of real world hardware backdoors may make it easier to defend against them.

References

1. S. Adee. The hunt for the kill switch. *IEEE Spectrum Magazine*, 45(5):3439, 2008.

2. 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 8797, New York, NY, USA, 2000. ACM.

3. M. Hicks, M. Finnicum, S. T. King, M. K. Martin and J. M. Smith. Overcoming an untrusted computing base: detecting and removing malicious hardware automatically. *IEEE Security and Privacy*, 2010.

4. King, Tucek, Cozzie, Grier, Jiang, Zhou. Designing and Implementing Malicious Hardware *Proceedings of the 1st USENIX Workshop and Large-Scale Exploits and Emergent Threats pp. 1-8 2008.*

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

6. *The SPARC Architecture Manual, Version 8.* Menlo Park, CA, 1992.

7. Sun Microsystems. *The SuperSPARC Microprocessor, Technical White Paper.* 1992.

8. A. Waksman and S. Sethumadhavan. Tamper Evident Microprocessors. In *Proceedings of the 31st IEEE Symposium on Security & Privacy* (Oakland), May 2010.