Network-on-Chip Firewall: Countering Defective and Malicious System-on-Chip Hardware

Michael LeMay[⋆] and Carl A. Gunter

University of Illinois at Urbana-Champaign

Abstract. Mobile devices are in roles where the integrity and confidentiality of their apps and data are of paramount importance. They usually contain a System-on-Chip (SoC), which integrates microprocessors and peripheral Intellectual Property (IP) connected by a Network-on-Chip (NoC). Malicious IP or software could compromise critical data. Some types of attacks can be blocked by controlling data transfers on the NoC using Memory Management Units (MMUs) and other access control mechanisms. However, commodity processors do not provide strong assurances regarding the correctness of such mechanisms, and it is challenging to verify that all access control mechanisms in the system are correctly configured. We propose a NoC Firewall (NoCF) that provides a single locus of control and is amenable to formal analysis. We demonstrate an initial analysis of its ability to resist malformed NoC commands, which we believe is the first effort to detect vulnerabilities that arise from NoC protocol violations perpetrated by erroneous or malicious IP.

1 Introduction

Personally administered mobile devices are being used or considered for banking, business, military, and healthcare applications where integrity and confidentiality are of paramount importance. The practice of dedicating an entire centrally administered phone to each of these apps is being abandoned in favor of granting access to enterprise data from personal devices as workers demand the sophistication available in the latest consumer mobile devices [6].

Security weaknesses of popular smartphone OSes have motivated isolation mechanisms for devices handling critical data, including hypervisors that operate at a lower level within the system [20]. For example, hypervisors can isolate a personal instance from a sensitive instance of Android, where both instances run simultaneously within Virtual Machines (VMs) on a single physical device. However, virtualized and non-virtualized systems both rely on the correctness of various hardware structures to enforce the memory access control policies that the system software specifies to enforce isolation.

Mobile devices are usually based on a System-on-Chip (SoC) containing microprocessor cores and peripherals connected by a Network-on-Chip (NoC).

[⋆] M. LeMay was with the University of Illinois at Urbana-Champaign while performing the work described herein, but he was employed by Intel Corporation at the time of submission. The views expressed are those of the authors only.

Each component on the SoC is referred to as an Intellectual Property (IP) core or block. A single SoC may contain IP originating from many different entities. SoC IP may be malicious intrinsically at the hardware level, or it may be used to perform an attack orchestrated by software, and such IP may lead to compromises of critical data. Such attacks would involve data transfers over the NoC. Memory Management Units (MMUs) and IO-MMUs can potentially prevent such attacks.

Commodity processors do not provide strong assurances that they correctly enforce memory access controls, but recent trends in system design may make it feasible to provide such assurances using enhanced hardware that is amenable to formal analysis. In this paper, we propose the hardware-based *Network-on-Chip Firewall (NoCF)* that we developed using a functional hardware description language, Bluespec. Bluespec is a product of Bluespec, Inc. Although Bluespec has semantics based on term-rewriting systems, those semantics also reflect characteristics of hardware [1]. We developed an embedding of Bluespec into Maude, which is a language and set of tools for analyzing term-rewriting systems. At a high level, term-rewriting systems involve the use of atomic rules to transform the state of a system. We know of no elegant way to directly express the hardware-specific aspects of Bluespec in a Maude term-rewriting theory, so we used Maude strategies to control the sequencing between rules in the theories to match the hardware semantics [16]. We then used our model to detect attacks that violate NoC port specifications, which have previously received little attention.

A lightweight processor core is dedicated to specifying the NoCF policy using a set of *policy configuration interconnects* to interposers, which provides a single locus of control. It also permits NoCF to be applied to NoCs lacking access to memory, avoids the need to reserve system memory for storing policies when that memory is available, and simplifies the internal logic of the interposers. The policy can be pre-installed or specified dynamically by some entity such as a hypervisor within the system. The interposers and associated policies are distributed to accommodate large NoCs.

To demonstrate one type of attack that can be blocked by NoCF, we construct a malicious IP block analogous to a Graphics Processing Unit (GPU) and show how it can be instructed to install a network keylogger by any app that simply has the ability to display graphics. This attack could be used to achieve realistic, malicious objectives. For example, a government seeking to oppress dissidents could convince them to view an image through a web browser or social networking app and subsequently record all of their keystrokes.

Our contributions include:

- An efficient, compact NoCF interposer design that is amenable to formal analysis and provides a single locus of control.
- An embedding of Bluespec into the Maude modeling language.
- Use of formal techniques to discover a new attack.
- A triple-core FPGA prototype that simultaneously runs two completely isolated, off-the-shelf instances of Linux with no hypervisor present on the cores or attached to the NoCs hosting Linux at runtime.

The rest of this paper is organized as follows. §2 provides background on SoC technology. §3 describes the threat model. §4 describes a core-based isolation approach. §5 discusses the design of NoCF interposers. §6 describes a NoCF prototype system. §7 discusses how NoCF can help to mitigate a sample attack. §8 formally analyzes the prototype. §9 discusses related work. §10 concludes the paper. Please refer to our technical report for additional details [14].

2 Background

Each block of IP on an SoC can be provided by an organization within the SoC vendor or by an external organization. SoCs commonly contain IP originating from up to hundreds of people in multiple organizations and spread across multiple countries [27]. Some IP (e.g. a CPU core) may be capable of executing software whereas other IP may only offer a more rudimentary configuration interface, e.g. one based on control registers. It is difficult to ensure that all of the IP is high-quality, let alone trustworthy [5, 9]. The general trend is towards large SoC vendors acquiring companies to bring IP development in-house [24]. However, even in-house IP may provide varying levels of assurance depending on the particular development practices and teams involved and the exact nature of the IP in question. For example, a cutting-edge, complex GPU may reasonably be expected to exhibit more errors than a relatively simple Wi-Fi controller that has been in use for several years. Furthermore, malicious hardware can be inserted at many points within the SoC design and manufacturing process and can exhibit a variety of behaviors to undermine the security assurances of the system [4]. Memory Management Units (MMUs) and IO-MMUs are commonly used to restrict the accesses from IP blocks, which can constrain the effects of erroneous or malicious IP. Thus, errors that can permit memory access control policies to be violated are the most concerning. A sample system topology is depicted in Figure 1. It shows two CPU cores, a GPU, a two-level interconnect, and some examples of peripherals. Note that the GPU is both an interconnect master and a peripheral.

An MMU is a component within a processor core that enforces memory access control policies specified in the form of page tables that are stored in main memory. Some SoCs incorporate IO-MMUs that similarly restrict and redirect peripheral master IP block NoC data transfers. A page table contains entries that are indexed by part of a virtual address and specify a physical address to which the virtual address should be mapped, permissions that restrict the accesses performed using virtual addresses mapped by that entry, whether the processor must be in privileged (supervisor) mode when the access is performed, and auxiliary data. Page tables are often arranged hierarchically in memory, necessitating multiple memory accesses to map a particular virtual address. To reduce the expense incurred by page table lookups, the MMU contains a Translation Lookaside Buffer (TLB) that caches page table entries in very fast memory inside the MMU. In the case of an MMU, each isolated software component (such as a process or VM) is typically assigned a dedicated page table. Correspondingly for an IO-MMU,

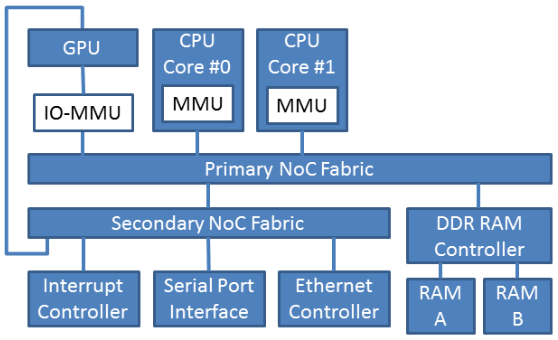


Fig. 1: Example SoC system topology.

one or more page tables may be assigned to each device. By only mapping a particular region of physical memory in one of the component's page tables, that memory is protected from accesses by other components. The relatively high complexity of modern MMUs and IO-MMUs increases the likelihood of errors that undermine their access control assurances [10]. Since NoCF does not use page tables nor does it provide address translation support, it is much less complex and can constrain an attack leveraging a vulnerable MMU or IO-MMU. Note that these technologies are not mutually exclusive. In fact, it is useful to provide defense-in-depth by enforcing coarse partitions with NoCF and relying on MMUs and IO-MMUs to implement finer-grained controls within each partition.

It could be preferable to formally verify existing MMUs and IO-MMUs rather than devising new protection mechanisms. However, it is challenging to formally verify MMUs and IO-MMUs. Formal verification techniques can prove the absence of design errors within commercial processor cores, but they currently only provide a good return-on-investment when used instead to detect errors [3]. To the best of our knowledge, MMUs have only been formally verified in experimental processors [8,22]. The policy data for MMUs and IO-MMUs is itself protected by them, so it is likely to be more challenging to verify that the policy is trustworthy compared to the NoCF policy implemented on an isolated core. Finally, the MMU is a central part of each processor core with many interfaces to other parts of the core, complicating analysis. We have not formally verified NoCF either, but we demonstrate how to develop a model of it that is amenable to formal analysis. This is a non-trivial precondition for formal verification.

Individual blocks of IP communicate using one or more NoCs within a single SoC. A NoC is not simply a scaled-down network comparable to, e.g. an Ethernet LAN. Networks for large systems, such as LANs, have traditionally been connection-oriented, predominantly relying on protocols such as TCP/IP. Networks for small systems, such as NoCs, have traditionally lacked support for persistent connections. Older SoC designs relied on buses, which are subtly distinct from NoCs. For our purposes, it is not necessary to distinguish between buses and NoCs. We are concerned primarily with their external ports, which are common between both types of interconnects. Slave devices accessible over a NoC are assigned physical address ranges, so memory access controls like those in NoCF can also be used to control access to devices.

The protection mechanisms that we propose are inserted between the NoC and the IP, and they do not necessitate changes to individual IP blocks. Thus, NoCF could be added quite late in the design process for an SoC, after the main functionality of the SoC has been implemented.

3 Threat Model

Software running on a particular core is assumed to be arbitrarily malicious and must be prevented from compromising the confidentiality, integrity, and availability of software on other cores. The system software that configures NoCF must correctly specify a policy to enforce isolation between the cores. Recent work on minimizing the Trusted Computing Base (TCB) of hypervisors and formally verifying them may be helpful in satisfying this requirement [13, 25].

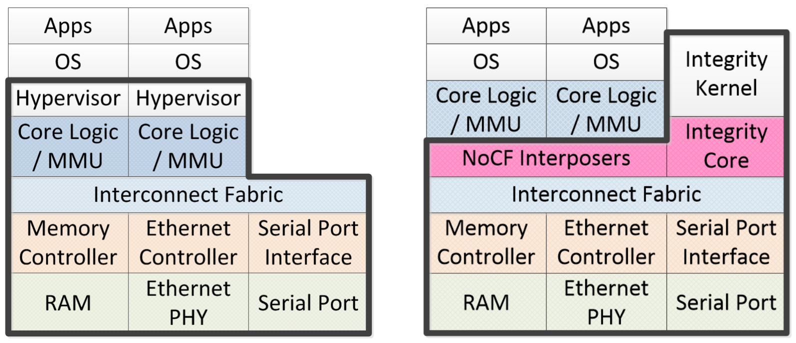
Our concern in this paper is that isolation between cores that are protected in this manner could potentially be compromised by misbehaving IP. Note that the memory controller in Figure 1 is connected to two Random Access Memories (RAMs). For the purpose of the threat model discussion, the data in RAM A should only be accessible to the GPU and CPU Core #0 and RAM B should only be accessible to CPU Core #1. We now define the types of compromises we seek to prevent:

1. *Confidentiality:* Some misbehaving IP may construct an unauthorized information flow from some other target IP transferring data that the misbehaving IP or the VM controlling it is not authorized to receive. This flow may be constructed with or without the cooperation of the target IP. The misbehaving IP may have authorization to access a portion of the target IP, but not the portion containing the confidential data. For example, CPU Core #0 or the GPU could potentially transfer data from RAM B to RAM A, since both CPU cores and the GPU have access to the shared memory controller. As another example, a misbehaving memory controller itself could perform that transfer.
2. *Integrity:* Some misbehaving IP may unilaterally construct an unauthorized information flow to other target IP to corrupt data. For example, the GPU or CPU Core #0 may modify executable code or medical sensor data in RAM B.
3. *Availability:* Resource sharing is an intrinsic characteristic of SoCs, so there is the possibility that misbehaving IP may interfere with other IP using those shared resources. For example, the GPU or CPU Core #0 could flood the NoC with requests to monopolize the NoC and interfere with NoC requests from CPU Core #1.

IP can manipulate wires that form its NoC port in an arbitrary manner. The IP might not respect the port clock and can perform intra-clock cycle wire manipulations. The IP might also violate the protocol specification for the port.

Since NoCF performs address-based access control, the NoC fabric is assumed to be trusted to selectively and accurately route requests and responses to and from the appropriate IP to prevent eavesdropping and interference from other IP cores. The integrity core expects each peripherals to be associated with particular ranges of addresses and each master IP core to be associated with particular NoC ports, and it uses that information to configure NoCF policies. Thus, a necessary condition for the correct enforcement of the security policy intended by the integrity core is that the NoC fabric operate in a trustworthy manner. Establishing trust in NoC fabrics is an orthogonal research issue.

We assume that slave devices are trusted to correctly process requests. For example, the memory controller must properly process addresses that it receives to enforce policies that grant different IP access to different regions of a memory accessible through a single shared memory controller. Establishing trust in such devices is an orthogonal research issue.



(a) Unaltered, hypervisor-based system.

(b) Similar system protected by NoCF.

Fig. 2: Comparison of TCBs, which are within the thick lines. Colored areas depict layers of hardware.

Covert channels are more prevalent between components that have a high degree of resource sharing, such as between software that shares a processor cache. Thus, NoCF provides tools to limit covert channels by restricting resource sharing. However, we do not attempt to eliminate covert channels in this work.

A mobile device may be affected by radiation and other environmental influences that cause unpredictable modifications of internal system state. A variety of approaches can handle such events and are complementary to our effort to handle misbehaviors that arise from the design of the device [19].

System software could be maliciously altered so that even if the intended NoC access control policy is correctly enforced, some overarching system security objective may still be violated. The operation of each IP core is determined not only by its hardware design and its connectivity to other IP cores, but also by how it is configured at runtime and what software it executes (if applicable). Trusted computing techniques can defeat attacks that alter system software by ensuring that only specific system software is allowed to execute [2]. We focus on techniques whereby the SoC vendor can constrain untrustworthy IP in its chip designs. Software security and hardware tamper-resistance techniques can further improve assurances of overall system security by checking for the correct operation of trusted IP.

4 Core-Based Isolation

Assigning software components to separate cores eliminates vulnerabilities stemming from shared resources such as registers and L1 caches. Regulating their activities on NoCs with a dynamic policy addresses vulnerabilities from sharing main memory or peripherals. We initially focus on isolating complete OS instances, since the memory access control policies required to accomplish that are straightforward and coarse-grained. However, NoCF could also be used to implement other types of policies.

The NoCF policy either needs to be predetermined or defined by a hypervisor, like the hypervisor specifies MMU policies. The policy will be maintained by an *integrity kernel* that runs on a dedicated *integrity core*, which will be discussed further below. The effect that this has on the TCB of a system with minimal resource sharing, such as our prototype that isolates two Linux instances, is depicted in Figure 1. The TCB will vary depending on how the policy is defined, since any software that can influence the policy is part of the TCB. In this example, the policy that was originally defined in a hypervisor is now defined in the integrity kernel, completely eliminating the hypervisor.

NoCF provides a coarser and more trustworthy level of memory protection in addition to that of the MMU and IO-MMU. These differing mechanisms can be used together to implement trade-offs between isolation assurances and costs stemming from an increased number of cores and related infrastructure.

The integrity core must be able to install policies in NoCF interposers and must have sufficient connectivity to receive policy information from any other system entities that are permitted to influence policies, such as a hypervisor. It may be possible to place the integrity kernel in firmware with no capability to communicate with the rest of the system, if a fixed resource allocation is desired. On the other end of the spectrum of possible designs, the integrity core may have full access to main memory, so it can arbitrarily inspect and modify system state. Alternately, it may have a narrow communication channel to a hypervisor. Placing the integrity kernel on an isolated integrity core permits the pair of them to be analyzed separately from the rest of the system. However, it is also possible to assign the role of integrity core to a main processor core to reduce hardware resource utilization, even if the core is running other code.

5 NoCF Interposer Design

We now discuss the design decisions underlying NoCF. We base our design on the widely-used AMBA AXI4 NoC port standard. The rule format and storage mechanism of the Policy Decision Point (PDP) are loosely modeled after those of a TLB. The PDP decides which accesses should be permitted so that a Policy Enforcement Point (PEP) can enforce those decisions. Policy rules are inserted directly into the PDP using a policy configuration interconnect to an integrity core. This reduces the TCB of the PDP relative to a possible alternate design that retrieves rules from memory like an MMU. The integrity core is a dedicated, lightweight processor core that is isolated from the rest of the system to help protect it from attack. The decisions from the PDP are enforced for each address request channel by that channel's PEP.

The AXI4 specification defines two matching port types. The master port issues requests and the slave port responds to those requests. Each pair of ports has two distinct channels, one for read requests and one for write requests. This port architecture enables us to easily insert NoCF interposers, each of which contains a PDP, an integrity core interface, and two PEPs, one for each channel. Each interposer provides both a master and slave port so that it can be interposed between each IP master port and the NoC slave port that it connects to. A single interposer is depicted in Figure 3.

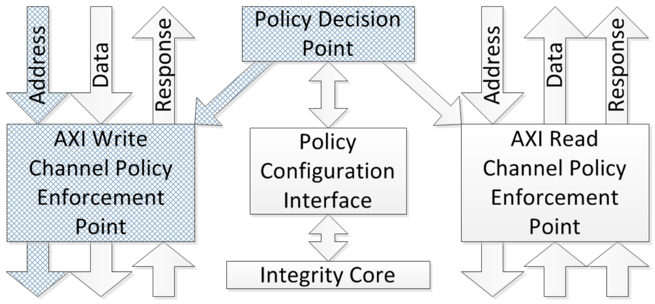

Address
Data
Response
Policy Decision Point
Address
Data
Response
AXI Write Channel Policy Enforcement Point
Policy Configuration Interface
AXI Read Channel Policy Enforcement Point
Integrity Core

Fig. 3: Internal configuration of NoCF interposer. Each interposer contains all of these components. Hatched regions are formally analyzed in §8.

We evaluate our design in a prototype system containing two main processor cores in addition to the integrity core, plus a malicious GPU. We now consider it as a sample system arrangement, although many other system arrangements are possible. Each main core has four AXI4 master ports. They connect to two NoCs in the system, one of which solely provides access to the DDR3 main memory controller, while the other connects to the other system peripherals. Each main core has two ports connected to each NoC, one each for instruction and data accesses. The GPU has a single master port connected to the NoC with the main memory, along with a slave port connected to the peripheral NoC (not shown).

We depict this topology in Figure 4. One interposer is assigned to each of the ports between the master IP and the NoCs, with a corresponding policy configuration interconnect to the integrity core. The depicted interconnect topology is slightly simplified compared to the one used in the commercial ARM Cortex-A9 MP processor, which shares an L2 cache between up to four cores. Thus, it would be necessary in that processor to place interposers between the cores and the L2 cache controller and to trust that controller to implement memory addressing correctly.

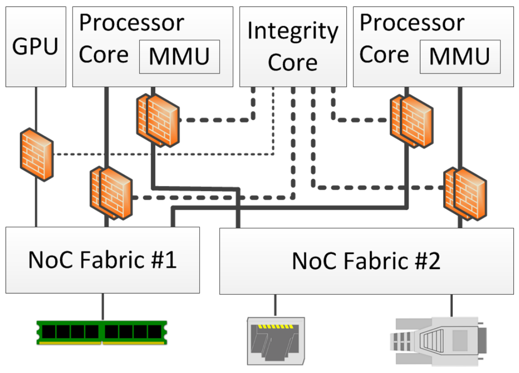


Fig. 4: System topology. A brick wall represents a NoCF interposer on one NoC port. Dashed lines denote policy configuration interconnects. Solid lines denote NoC ports. For interconnects and ports, thin lines denote single items and thick lines denote pairs.

Since NoCF interposers are distributed, they can each use a policy tailored to the port being regulated and this also concentrates the internal interfaces containing many wires between the PDP and PEPs in a small area of the chip while using an interface containing few wires to span the potentially long distance to the integrity core. However, it may be useful in some cases to share a PDP between several interposers that are subject to a single policy. That approach would reduce the number of policy configuration interconnects and the total PDP policy storage. A more complex approach would be to support selectively-shared rules for separate interposers in a single PDP, which would still reduce interconnect logic and could provide some reduction in PDP storage.

Each policy rule specifies a region of memory to which read and/or write access is permitted. A region is defined by a base address and a mask length specifier, which indicates the size of the region as one of a set of possible powers of two. This type of policy can be implemented very efficiently in hardware and corresponds closely to the policies defined by conventional MMU page tables.

Address requests are regulated by PEPs in cooperation with the PDP. The PDP stores a fixed number of rules in its database. The PDP checks the address in the request against all policy rules in parallel. Whenever a request matches some rule that has the appropriate read or write permission bit set, it will be permitted to pass through the PEP. Otherwise, the PDP sends an interrupt to the integrity core and also sends it information about the failing request. It then blocks the request until the integrity core instructs it to resume.

The integrity core can modify the policy rules prior to issuing the resume command. To modify policy rules, the integrity core sends commands over the policy configuration interconnect to insert a policy rule or flush all existing policy rules. Other commands could be defined in the future. When the interposer receives the resume command, it re-checks the request and either forwards it if it now matches some rule, or drops it and returns an error response to the master. It could also do something more drastic, such as blocking the clock signal or power lines feeding the master that issued the bad request.

Addresses other than the one in the request may be accessed during the ensuing data transfer. A variety of addressing modes are supported by AXI4 that permit access to many bytes in a burst of data transfers initiated by a single request. The policy administrator must account for these complexities by ensuring that all bytes that can actually be accessed should be accessible.

It can be useful to physically separate a protection mechanism from the surrounding logic and constrain its interfaces to that logic so that it can be independently analyzed [11]. This is possible in the case of NoCF, since its only interfaces are the controlled NoC ports and the policy configuration interconnect.

The PDP, integrity core interface, and PEP are all implemented in Bluespec to leverage its elegant semantics and concision, with interface logic to the rest of the prototype hardware system written in Verilog and VHDL. For details, please see our technical report [14].

6 Prototype Implementation

We used a Xilinx ML605 evaluation board, which includes a Virtex-6 FPGA, to implement a prototype of NoCF. We use MicroBlaze architecture processor cores implemented by Xilinx, because they are well-supported by Xilinx tools and Linux. The integrity core is very lightweight, with no cache, MMU, or superfluous optional instructions. It is equipped with a 16KiB block of on-chip RAM directly and exclusively connected to the instruction and data memory ports on the integrity core. This RAM is thus inaccessible from the other cores.

The prototype runs Linux 3.1.0-rc2 on both main cores, including support for a serial console from each core and exclusive Ethernet access from the first core. We compiled the Linux kernel using two configurations corresponding to each core so that they use different regions of system memory and different sets of peripherals. This means that no hypervisor beyond the integrity kernel is required, because the instances are completely separated. The system images are loaded directly into RAM using a debugger.

The integrity kernel specifies a policy that constrains each Linux instance to the minimal memory regions that are required to grant access to the memory and peripherals allocated to the instance. Attempts to access addresses outside of an instance's assigned memory regions cause the instance to crash with a bus error, which is the same behavior exhibited by a system with or without NoCF when an instance attempts to access non-existent physical memory.

The interposers each contain two policy rules and replace them in FIFO order, except that the interposers for data loads and stores to the peripherals contain four policy rules each, since they are configured to regulate fine-grained memory regions.

7 Constraining a Malicious GPU

A malicious GPU could perform powerful attacks, since it would have bus-master access and be accessible from all apps on popular mobile OSes. Almost all apps have a legitimate need to display graphics, so software protection mechanisms that analyze app behavior could not be expected to flag communications with the GPU as suspicious, nor could permission-based controls be configured to block such access.

We constructed hardware IP that is analogous to a hypothetical malicious GPU. It has both master and slave AXI4 interfaces. In response to commands received on its slave interface, the IP reads data from a specified location in physical memory. This is analogous to reading a framebuffer. The IP inspects the least significant byte of each pixel at the beginning of the framebuffer. This is a very basic form of steganographic encoding that only affects the value of a single color in the pixel, to reduce the chance of an alert user visually detecting the embedded data. More effective steganographic techniques could easily be devised. If those bytes have a specific "trigger" value, then the IP knows that part of the framebuffer contains a malicious command. The trigger value is selected so that it is unlikely to appear in normal images. The IP then continues reading steganographically-embedded data from the image and interprets it as a command to write arbitrary data embedded in the image to an arbitrary location in physical memory.

We developed a simple network keylogger to be injected using the malicious IP. The target Linux system receives user input via a serial console, so the keylogger modifies the interrupt service routine for the serial port to invoke the main keylogger routine after retrieving each character from the serial port hardware. This 20 byte hook is injected over a piece of error-checking code that is not activated in the absence of errors. The physical address and content of this error-checking code must be known to the attacker, so that the injected code can gracefully seize control and later resume normal execution. The keylogger hook is generated from a short assembly language routine.

The main keylogger routine is 360 bytes long and sends each keystroke as a UDP packet to a hardcoded IP address. It uses the optional netpoll API in the Linux kernel to accomplish this in such a compact payload. This routine is generated from C code that is compiled by the attacker as though it is a part of the target kernel. The attacker must know the addresses of the relevant netpoll routines as well as the address of a region of kernel memory that is unused, so that the keylogger can be injected into that region without interfering with the system's business functions. We chose a region pertaining to NFS functionality. The NFS functionality was compiled into the kernel, but is never used on this particular system.

All of the knowledge that we identified as being necessary to the attacker could reasonably be obtained if the target system is using a standard Linux distribution with a known kernel and if the attacker knows which portion of the kernel is unlikely to be used by the target system based on its purpose. For example, other systems may use NFS, in which case it would be necessary to find a different portion of the kernel that is unused on that system in which to store the keylogger payload.

To constrain the GPU in such a way that this attack fails, it is simply necessary to modify the NoCF policy to only permit accesses from the GPU to its designated framebuffer in main memory, as is depicted in Figure 5.

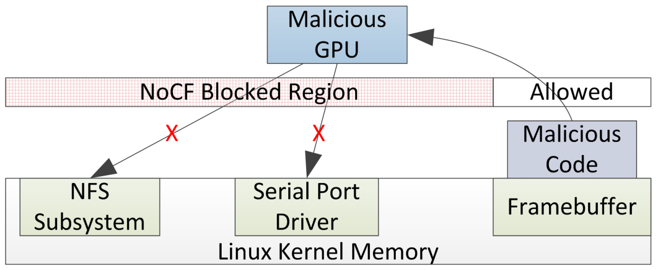


Fig. 5: NoCF can be configured to block attacks that rely on writes by malicious hardware to specific memory locations that it has no legitimate need to access.

This particular attack could also be blocked by a kernel integrity monitor, which ensures that only approved kernel code is permitted to execute [23]. The malware injected by the GPU would not be approved, so it would be unable to execute. However, kernel integrity monitors fail to address attacks on userspace and can be complex, invasive, and high-overhead.

8 Formal Analysis

8.1 Analysis Overview

We developed a shallow embedding of a subset of Bluespec into Maude, a native term rewriting system, and used a Maude model of NoCF to precisely identify a subtle vulnerability in NoCF. A shallow embedding is one where source terms are mapped to target terms whose semantics are natively provided by the target system. We only model the portion of the system that is shown with a hatched background in Figure 3. This model was sufficient to detect an interesting vulnerability, although a complete model would be necessary to analyze the entire NoCF system in the future.

We manually converted substantial portions of the Bluespec code for NoCF to Maude using a straightforward syntactic translation method that could be automated. The hardware design is described in our technical report [14]. We developed our Bluespec description with no special regard for its amenability to analysis, so the subset of the Bluespec syntax that we modeled has not been artificially restricted. We modeled each variable name and value for Bluespec structures, enumerations, and typedefs as a Maude term. We defined separate sorts for variable names and for data values that can be placed in Bluespec registers or wires. We defined subsorts for specific types of register data, such as the types of data that are transferred through AXI interfaces and the state values for each channel. We defined a separate sort for policy rules.

The model was structured as an object-oriented system. Several distinct message types can be sent between objects. All of them specify a method to be invoked, or that was previously invoked and is now returning a value to its caller. Anonymous and return-addressed messages are both supported. The latter specify the originator of the message. These are used to invoke methods that return some value. There are staged variants of the anonymous and return-addressed message types that include a natural number indicating the stage of processing for the message. This permits multiple rewrite rules to sequentially participate in the processing of a single logical message. Return messages wrap some other message that was used to invoke the method that is returning. They attach an additional piece of content, the return value, to the wrapped message. Special read and write token messages regulate the model's execution, as will be described below. Finally, two special types of messages are defined to model interactions over the FSL interface. An undecided message contains an address request, modeling an interposer notifying the integrity core of a blocked request. An enforce write message models the integrity core instructing the interposer to recheck the blocked request. Those two message types abstract away the details of FSL communication, since those are not relevant to the Critical Security Invariant described below.

We defined equations to construct objects modeling the initial state of each part of the system. We defined Maude object IDs as hierarchical lists of names to associate variables with the specific subsystem in which they are contained and to represent the hierarchical relationships between subsystems. We defined five classes corresponding to the Bluespec types of variables in the model. Registers persistently store some value. The value that was last written into the register in some clock cycle prior to the current one is the value that can be read. A register always contains some value. Wires can optionally store some value within a single clock cycle. Pulse wires behave like ordinary wires, but they can only store a single unary value. OR pulse wires behave like pulse wires, but it is possible for them to be driven multiple times within a single clock cycle. They will only store a unary value if they are driven at least once during the clock cycle.

We modeled Bluespec methods as rewrite rules. The required activation state for the relevant objects is written on the left hand side of the rule, and the transformed state of those objects is written on the right hand side. Either side can contain Maude variables. Simple Bluespec rule conditions can be represented by embedding the required variable values into the left hand side of the corresponding Maude rule. More complex conditions can be handled by defining a conditional Maude rule that evaluates variables from the left hand side of the Maude rule. Updates to register variables require special handling in Maude. We define a wire to store the value to be written to the register prior to the next clock cycle, and include a Maude rewriting rule to copy that value into the register before transitioning to the next cycle.

We modeled Bluespec functions as Maude equations. We also defined Maude functions to model complex portions of Bluespec rules, such as a conditional expression.

The main challenge that we overcame in embedding Bluespec in Maude stems from the fact that Maude by default implements something similar to pure term rewriting system semantics, in which no explicit ordering is defined over the set of rewrite rules. To model the modified term rewriting semantics of Bluespec, we imposed an ordering on the rules in the Maude theory that correspond to Bluespec rules and restricted them to fire at most once per clock cycle. This includes rules to model the implicit Bluespec rules that reset ephemeral state between cycles. We used and extended the Maude strategy framework to control rule execution [16]. The Bluespec compiler output a total ordering of the rules that was logically equivalent to the actual, concurrent schedule it implemented in hardware. We applied that ordering to the corresponding Maude rules.

To model bit vectors, we relied on a theory that had already been developed as part of a project to model the semantics of Verilog in Maude [17].

To search for vulnerabilities in NoCF, we focused on the following Critical Security Invariant:

Invariant 1 *If an address request is forwarded by a NoCF interposer, then it is permitted by the policy within that interposer.*

We modeled some basic attack behaviors to search for ways in which that invariant could be violated. In particular, we specified that during each clock cycle attackers may issue either a permissible or impermissible address request, relative to a predefined policy, or no address request. The AMBA AXI4 specification requires master IP to wait until its requests have been acknowledged by the slave IP before modifying them in any way, but our model considers the possibility that malicious IP could violate that.

We used the Maude "fair rewriting" command to perform a breadth-first search of the model's possible states for violations of the Critical Security Invariant. We extended the Maude strategy framework to trace the rule invocations so that the vulnerabilities underlying detected attacks could be independently verified and remedied.

8.2 Formalization Details

We extended each solution term with an ordered list of quoted identifiers identifying the rules that were invoked to reach the solution. Correspondingly, we extended each task term with a list of quoted identifiers, so that the extended signature of the primary task constructor is as follows:

```
op <_@_via_> : Strat Term QidList -> Task .
```

We extended the equations and rules for evaluating strategies that were necessary for our particular model to also propagate and modify the list of invoked rules.

We defined an object-based model, so we adapted the Maude strategy framework to encapsulate a Maude term of sort `Configuration` directly in each solution term to enhance readability of the output [7]. We marked the solution terms as frozen to prevent any further transformation of the encapsulated configuration term. We used Maude reflection to transform the representation of the configuration term being manipulated by the strategy framework when a solution is recorded. For example, consider our modified definition for the rule for the idle strategy:

```
vars T T' : Term .
var QL : QidList .
rl < idle @ T via QL > => sol-from(T, QL) .
```

For reference, the original definition was:

```
rl < idle @ T > => sol(T) .
```

Analogous modifications were made to other rules in the strategy framework. We defined `sol-from` as follows:

```
var CNF : Configuration .
op sol-from : Term QidList -> Task .
eq sol-from(T, QL) = sol(downTerm(T, err-cnf), QL) .
```

Note that this relies on a term of kind `Configuration` we defined to represent an error, `err-cnf`.

The reverse transformation is needed in the concatenation rules and other rules that need to manipulate the configuration term in the solution, such as:

```
var TASKS : Tasks .
var E : Strat .
eq < sol(CNF, QL) TASKS ; seq(E) > =
  < E @ upTerm(CNF) via QL > < TASKS ; seq(E) > .
```

The rule application equations extend the list of invoked rules, as in the following:

```
var L : Qid .
var Sb : Substitution .
var N : Nat .
var Ty : Type .
ceq apply-top(L, Sb, T, N, QL) =
  sol-from(T', QL L) apply-top(L, Sb, T, N + 1, QL)
    if { T', Ty, Sb' } := metaApply(MOD, T, L, Sb, N) .
```

We now describe the specific strategy used for model checking the Critical Security Invariant. The following defines the initial state for model checking. `mkNoCF` is defined elsewhere to initialize the configuration term for the main NoCF model, and `read-tok` and `write-tok` are the read and write tokens, resp., that were described previously.

```
op init-stt : -> Configuration .
eq init-stt = mkNoCF(nocf) read-tok write-tok .
```

The following represents an abbreviated list of the methods and rules *in the Bluespec model* in the order in which they should be invoked, as specified by the Bluespec compiler:

```
sort FireRule .
ops mandatory-fr fr : NeBluespecIdList Qid -> FireRule .

op bluespec-sched : -> NeFireRuleList .
eq bluespec-sched =
  fr(nocf wrtAddrFilter, 'in_issue)
  fr(nocf wrtAddrFilter, 'ready)
  ...
  fr(nocf, 'wrt_addr_resume)
  fr(nocf, 'upd_wrt_stt)
  ...
  fr(nocf wrtAddrFilter, 'clear)
  mandatory-fr(nocf, 'nocf_clear) .
```

The first parameter for each `FireRule`, a term of sort `NeBluespecIdList`, is a list of identifier terms that identifies an object in a hierarchical Bluespec design relative to which the modeled Bluespec method or rule should be invoked. The second parameter is the name of the Maude rule that models the corresponding Bluespec method or rule. The objects in the configuration term use these same hierarchical identifiers prepended to individual variable identifers. The distinction between `fr` and `mandatory-fr` rules is that `fr` rules are invoked iff they can possibly be applied and strategy execution continues regardless whereas `mandatory-fr` rules are always invoked. We use mandatory invocation only at the end of the schedule list for a special rule that models the Bluespec behavior of clearing certain variables at the end of a hardware clock cycle. It is a conditional rule that is only enabled if a NoC request has not been finally denied. Thus, if a NoC request has been finally denied, then the rule is disabled and strategy execution terminates for that branch of the model state space.

The rules below convert the concise list of ordered Bluespec rules into a Maude strategy to enforce the desired ordering of rule invocation. They also bind the `PFX` variable that is used in the Maude rule identified by `Q` to the specified identifier, so that the Maude rule is applied specifically to the Bluespec object with that identifier.

```
op bluespec-strat : -> Strat .
eq bluespec-strat = sched-to-strat(bluespec-sched) .

op sched-to-strat : FireRuleList -> Strat .
eq sched-to-strat(fr(BI:NeBluespecIdList, Q:Qid)) =
  try(Q:Qid['PFX:NeBluespecIdList <-
    upTerm(BI:NeBluespecIdList)]) .
eq sched-to-strat(mandatory-fr(BI:NeBluespecIdList, Q:Qid)) =
  Q:Qid['PFX:NeBluespecIdList <- upTerm(BI:NeBluespecIdList)] .
eq sched-to-strat(FR:FireRule FRL:FireRuleList) =
  sched-to-strat(FR:FireRule) ;
  sched-to-strat(FRL:FireRuleList) [owise] .
```

The following strategy fragment may install zero or one out of a set of two policy rules to the NoCF interposer:

```
op add-policy : -> Strat .
eq add-policy =
  try('add_pol_rule_1[none]) | try('add_pol_rule_2[none]) .
```

The `amatch` test below searches each state term for any subterm that matches the term provided as the first parameter such that the condition in the second parameter is satisfied.

```
op bad-match : -> Strat .

eq bad-match = amatch('_~>_['_<`{_`}-_['PFX:NeBluespecIdList,
  'PFX1:NeBluespecIdList,'out-read.Method],'RD:RegData],
    '_==_['RD:RegData,upTerm(good-addr-req)] = 'false.Bool) .
```

The first parameter is a term representing a return message in our model. Taking it down a level in reflection makes the syntax more clear:

```
(PFX:NeBluespecIdList <{PFX1:NeBluespecIdList}- out-read)
  ~> RD:RegData
```

`PFX` is the object to which the original method call was directed, `PFX1` is the object that invoked the method and to which this return message is directed, `out-read` is the identifier of a method that was invoked with no parameters, and `RD` is the returned term resulting from modeling the method's execution. The `out-read` method models the interposer passing an approved address request along to the NoC. The condition in the `amatch` test checks for any such approved request that is not the single request that we defined to be allowed by policy in our Maude model. Thus, this test checks for any address requests that violate policy and yet are still (incorrectly) approved by the NoCF interposer. The Critical Security Invariant specifies that this test should never be satisfied.

The terms below define the initial state and strategy for the model checker.

```
op issue-addr-req : -> Strat .
eq issue-addr-req =
  ('issue_good[none]) | ('issue_bad[none]) | idle .

op init-term : -> Term .
op init-strat : -> Strat .
op init-task : -> Task .

eq init-term = upTerm(init-stt) .
eq init-strat =
  (((('issue_read[none]) | idle) ; (issue-addr-req) ;
  (add-policy) ; (bluespec-strat)) *) ; bad-match  .
eq init-task = < init-strat @ init-term via nil > .
```

The strategy is simply a loop representing zero or more hardware clock cycles of the device being modeled, followed by the test for violations of the Critical Security Invariant. The model can inject stimuli at the beginning of each clock cycle. The `issue_read` rule consumes the read token and updates the state to cause any currently approved address request from the NoCF interposer to be read out in that clock cycle, or any address request that is approved in a future clock cycle to be read out as soon as possible after that approval. This non-determinism models the fact that the NoC may not be immediately ready to accept an approved address request in the same clock cycle that the NoCF interposer approves it. The `issue_good` and `issue_bad` rules likewise consume the write token and issue an address request that either complies with or violates the policy, respectively, to the NoCF interposer. The possibility that the `idle` strategy may be selected instead of issuing an address request models the fact that address requests may or may not be issued every clock cycle in the hardware design. A rule is in place to reintroduce a new write token after an address request has been issued so that zero or one address requests can be issued during each clock cycle. Finally, fair rewriting is used to check the model represented by `init-task`. Solutions represent counterexamples to the Critical Security Invariant.

8.3 Analysis Results

We detected a subtle possible attack applicable to a straightforward implementation of the interposer. First, the attacker issues the permissible request, when the slave IP is not yet ready to accept a new request. The attacker then issues the impermissible request after the PEP has approved the first request and is simply waiting for the slave IP to accept the request. The PEP assumes the master adheres to the protocol specification and will wait for the initial request to be acknowledged, so it passes the request through from the master. This attack is depicted in Figure 6. This type of model is powerful, since it is a simple matter to model basic attacker behaviors which can then be automatically analyzed to detect complex attacks.

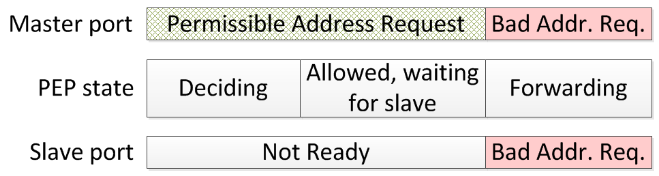


Fig. 6: Timing of address requests at port relative to PEP state for an attack forwarding an unchecked address request.

We implemented a countermeasure in the Bluespec code to block this attack. It now buffers the request that is subjected to access control checking, and then issues that exact request to the slave if it is allowed, regardless of the current state of the request interface from the master. This countermeasure introduces additional space overhead, so it is not something that a designer would reasonably be expected to include at the outset in a straightforward implementation.

8.4 Analysis Discussion

We extended the Verilog specification of the NoC ports to interface with the Verilog generated by the Bluespec compiler. This implies that we must trust or verify the Verilog interface code. The interface code consists almost entirely of straightforward wire connections, so it should be amenable to manual or formal analysis. We must also trust the Bluespec compiler to output Verilog code corresponding to the input Bluespec code.

Our threat model allows malicious IP to perform intra-clock cycle manipulations of the wires in the port to the interposer. The effects of such manipulations are difficult to analyze at the level of abstraction considered in this section. If this type of behavior is a concern, it can be easily suppressed by buffering the port using a slice of registers that effectively forces the IP to commit to a single port state for a full clock cycle from the perspective of the interposer. The commercial NoC IP that we used in our prototype supports the creation of such register slices with a simple design parameter. This solution would introduce a single clock cycle delay at the port.

Ultimately, NoCF and other elements of the TCB should be formally verified to be resistant to foreseeable attack types, and the analysis described here suggests that the elegant semantics of Bluespec helps to make such an effort more tractable than it would be if we had used Verilog or VHDL.

As an intermediate goal, it will be important to model more potential attacker behaviors to potentially identify additional vulnerabilities and formally verify the absence of vulnerabilities when possible. The model should be expanded to model all possible sequences of values that misbehaving IP could inject into the NoC ports to which it has access. A challenge is that each master controls a large number of input wires that feed into the NoC. Many of these wires carry 32-bit addresses and data, so inductive proof strategies may permit that number to be substantially reduced by showing that a model using narrower address and data ports is equivalent to the full model in the context of interesting theorems. Similarly, induction may permit long sequences of identical input values or repetitive sequences of input values to be collapsed to shorter sequences, if in fact the NoC logic does not impart significance to the patterns in question.

We considered one theorem for which we detected a counterexample, but there are many other theorems that are foundational to the system's trustworthiness and that should be used as guidance while analyzing NoCF. The process of identifying these theorems should be informed by past vulnerabilities, system requirements, and desirable information-theoretic properties.

Analyzing NoCF and the rest of the TCB with respect to the theorems and the detailed model we have proposed is an important step towards providing strong assurance that the system can be trusted to process sensitive data alongside potentially misbehaving hardware and software components. Our formal analysis of the existing NoCF prototype demonstrates the improved analysis capabilities that are enabled by formal hardware development practices and modern formal analysis tools. This suggests that the broader analysis effort we have proposed is feasible, given sufficient resources.

9 Related Work

In this section, we consider tools and techniques that enable formal reasoning about hardware. The primary novelty of NoCF is that it is a NoC access control mechanism designed to be amenable to formal analysis.

Advances have been made in languages for formally specifying information-flow properties in hardware like Caisson [15]. Tiwari et al. developed and verified an information-flow secure processor and microkernel, but that was not in the context of a mobile-phone SoC and involved radical modifications to the processor compared to those required by NoC-based security mechanisms [26]. Volpano proposed dividing memory accesses in time to limit covert channels [28]. Information-flow techniques could be generally applicable to help verify the security of the trusted components identified in §3.

Other techniques are complementary to these lines of advancement in that they offer approaches for satisfying the assumptions of our threat model. "Moats and Drawbridges" is the name of a technique for physically isolating components of an FPGA and connecting them through constrained interfaces so that they can be analyzed independently [11, 12].

SurfNoC schedules multiple protection domains onto NoC resources in such a way that non-interference between the domains can be verified at the gate level [29]. This could complement NoCF by preventing unauthorized communications channels between domains from being constructed in the NoC fabric.

Richards and Lester defined a shallow, monadic embedding of a subset of Bluespec into PVS and performed demonstrative proofs using the PVS theorem prover on a 50-line Bluespec design [21]. Their techniques may be complementary to our model checking approach for proving properties that are amenable to theorem proving. Katelman defined a deep embedding of BTRS into Maude [18]. BTRS is an intermediate language used by the Bluespec compiler. Our shallow embedding has the potential for higher performance, since we translate Bluespec rules into native Maude rules. Bluespec compilers could potentially output multiple BTRS representations for a single design, complicating verification. Finally, our embedding corresponds more closely to Bluespec code, which could make it easier to understand and respond to output from the verification tools.

10 Conclusion

Mobile devices that became popular for personal use are increasingly being relied upon to process sensitive data, but they are not sufficiently trustworthy to make such reliance prudent. Various software-based techniques are being developed to process data with different levels of sensitivity in a trustworthy manner, but they assume that the underlying hardware memory access control mechanisms are trustworthy. We discuss how to validate this assumption by introducing a NoC Firewall that is amenable to formal analysis. We present a prototype NoCF that is implemented using a hardware description language with elegant semantics. We demonstrate its utility by using it to completely isolate two Linux instances without running any hypervisor code on the cores hosting the instances.

Acknowledgments

This paper is dedicated to José Meseguer, whose work has inspired and formed the basis of so many studies like it.

The work was partially supported by HHS 90TR0003-01 (SHARPS) and NSF CNS 13-30491 (THaW). The views expressed are those of the authors only. We measured lines of code using David A. Wheeler's 'SLOCCount'.

References

1. Bluespec SystemVerilog overview. Tech. rep., Bluespec, Inc. (2006), `http://www.bluespec.com/products/documents/BluespecSystemVerilogOverview.pdf`
2. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: 18th IEEE Symposium on Security and Privacy. pp. 65–71. Oakland, CA, USA (May 1997)

3. Arditi, L.: Formal verification: So many applications. Design automation conference electronic chips & systems design initiative 2010 presentation, Anaheim, CA, USA (Jun 2010)
4. Beaumont, M., Hopkins, B., Newby, T.: Hardware trojans – prevention, detection, countermeasures (A literature review). Tech. Rep. DSTO-TN-1012, DSTO Defence Science and Technology Organisation, Edinburgh, South Australia (Jul 2011)
5. Butler, S.: Managing IP quality in the SoC era. Electronic Engineering Times Europe p. 5 (Oct 2011)
6. Cheng, R.: So you want to use your iPhone for work? Uh-oh. The Wall Street Journal (Apr 2011)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude manual (version 2.6). Tech. rep. (Jan 2011)
8. Dalinger, I.: Formal Verification of a Processor with Memory Management Units. Dr.-Ing., Saarland University, Saarbrücken, Germany (Jun 2006)
9. Goering, R.: Panelists discuss solutions to SoC IP integration challenges. Industry Insights - Cadence Community (May 2011)
10. Gotze, K.: A survey of frequently identified vulnerabilities in commercial computing semiconductors. In: 4th IEEE International Symposium on Hardware-Oriented Security and Trust. pp. 122–126. HOST, San Diego, CA, USA (Jun 2011)
11. Huffmire, T., Brotherton, B., Wang, G., Sherwood, T., Kastner, R., Levin, T., Nguyen, T., Irvine, C.: Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In: 28th IEEE Symposium on Security and Privacy. pp. 281–295. Oakland, CA, USA (May 2007)
12. Huffmire, T., Irvine, C., Nguyen, T.D., Levin, T., Kastner, R., Sherwood, T.: Handbook of FPGA Design Security. Springer (2010)
13. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: 22nd ACM Symposium on Operating Systems Principles. pp. 207–220. SOSP, Big Sky, MT, USA (Oct 2009)
14. LeMay, M., Gunter, C.A.: Network-on-Chip Firewall: Countering Defective and Malicious System-on-Chip Hardware (Apr 2014), `http://arxiv.org/abs/1404.3465`
15. Li, X., Tiwari, M., Oberg, J.K., Kashyap, V., Chong, F.T., Sherwood, T., Hardekopf, B.: Caisson: A hardware description language for secure information flow. In: 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 109–120. PLDI, San Jose, CA, USA (Jun 2011)
16. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for Maude strategies. In: 7th International Workshop on Rewriting Logic and its Applications. pp. 227–247. WRLA, Elsevier, Budapest, Hungary (Mar 2008)
17. Meredith, P., Katelman, M., Meseguer, J., Rosu, G.: A formal executable semantics of Verilog. In: 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign. pp. 179–188. MemoCODE, Grenoble, France (Jul 2010)
18. Michael Kahn Katelman: A Meta-Language for Functional Verification. Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois (2011)
19. Mukherjee, S.S., Emer, J., Reinhardt, S.K.: The soft error problem: An architectural perspective. In: 11th International Symposium on High-Performance Computer Architecture. pp. 243–247. HPCA, IEEE, San Francisco, CA, USA (Feb 2005)
20. Nachenberg, C.: A window into mobile device security: Examining the security approaches employed in Apple's iOS and Google's Android. Tech. rep., Symantec Security Response (Jun 2011)

21. Richards, D., Lester, D.: A monadic approach to automated reasoning for Bluespec SystemVerilog. Innovations in Systems and Software Engineering, Springer 7(2), 85–95 (Mar 2011)
22. Schubert, E.T., Levitt, K., Cohen, G.C.: Formal verification of a set of memory management units. Contractor Report 189566, National Aeronautics and Space Administration, Hampton, VA, USA (Mar 1992)
23. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: 21st ACM Symposium on Operating Systems Principles. pp. 335–350. SOSP, Stevenson, WA, USA (Oct 2007)
24. Shimpi, A.L.: NVIDIA to acquire Icera, adds software baseband to its portfolio. AnandTech.com (May 2011)
25. Szefer, J., Keller, E., Lee, R.B., Rexford, J.: Eliminating the hypervisor attack surface for a more secure cloud. In: 18th ACM Conference on Computer and Communications Security. CCS, Chicago, IL, USA (Oct 2011)
26. Tiwari, M., Oberg, J.K., Li, X., Valamehr, J., Levin, T., Hardekopf, B., Kastner, R., Chong, F.T., Sherwood, T.: Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In: 38th International Symposium on Computer Architecture. pp. 189–200. ISCA, ACM, San Jose, CA, USA (Jun 2011)
27. Villasenor, J.: Ensuring hardware cybersecurity. The Brookings Institution (May 2011)
28. Volpano, D.: Towards provable security for multilevel reconfigurable hardware. Tech. rep., Naval Postgraduate School (2008)
29. Wassel, H.M.G., Gao, Y., Oberg, J.K., Huffmire, T., Kastner, R., Chong, F.T., Sherwood, T.: SurfNoC: a low latency and provably non-interfering approach to secure networks-on-chip. In: 40th International Symposium on Computer Architecture. pp. 583–594. ISCA, ACM, Tel-Aviv, Israel (Jun 2013)