This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Eliminating Timing Information Flows in a Mix-trusted System-on-Chip

Jason Oberg*, Timothy Sherwood† and Ryan Kastner*
*Computer Science and Engineering, University of California, San Diego
{jkoberg,kastner}@cs.ucsd.edu

†Computer Science, University of California, Santa Barbara
sherwood@cs.ucsb.edu

***Abstract*—Modern computing systems continue to find themselves in control of applications which we rely on for our personal health and safety. These systems which require high-assurance have a very high-cost of failure. In order to build such a system with complete security, it must be built with a secure computing foundation. Creating such a secure hardware foundation is non-trivial for a number of reasons. One of which is due to the use of third-party intellectual property cores to reduce both the cost and design time of modern system-on-chips (SoCs) . Ensuring the integrity of trusted cores in these systems becomes difficult since the behavior of other untrusted cores is undefined. In this work, we show how information can be monitored at the level of Boolean gates to isolate trusted and untrusted cores in a modern SoC. We specifically target the Opencores WISHBONE cross-bar interconnect architecture and demonstrate how such isolation can be achieved. Further, we evaluate how effective the solution is by testing the system using a number of different scenarios.**

I. Introduction

Computing systems govern some of the most critical aspects of our lives. These high-assurance systems, which are found in medical devices, automobiles, planes, satellites, and military systems, have an extremely high cost of failure. Incorrect construction or unnoticed security holes can completely compromise their reliability, potentially putting humans in harms way of both their safety and privacy.

These systems have already seen their fair-share of security issues. For example, cardiac pacemakers have been shown to have weak radio-frequency (RF) security. This can be exploited to compromise both a patient's personal safety and their secrecy [1]. Aside from medical devices, security holes in automobiles have been exploited to show that many of the critical components (such as the braking system) can be remotely controlled by an attacker [2]. As is apparent with these examples, taking the utmost care in security when designing these systems is mandatory. However, in order to do so, designers need methods and tools which can help them expose security issues.

Some standards exist, such as the Common Criteria standard which specifies a set of rules in which secure systems must be constructed and evaluated. For example, the Evaluation Assurance Level (EAL) is awarded to systems based on how thoroughly they have been evaluated (assigned a number from 1 to 7). Not surprisingly, achieving a high-assurance level is not only time consuming but extremely expensive. Even further, it is nearly impossible to evaluate a system with components from untrusted entities since their behavior must be essentially assumed to be undefined. Since it is substantially faster and more cost efficient to use third-party components, it is desirable to construct a system which shows isolation between trusted, in-house built components, and potentially untrusted third party ones.

System-on-chips (SoCs) find themselves at the heart of these issues since they rely on the re-use of third-party intellectual property (IP) cores. These cores include memories, digital signal processors (DSP), graphical processing units (GPUs), analog RF blocks, I/O interfaces, and other various hardware accelerators (such as hardware encryption units). The SoC tightly integrates these cores together using a SoC bus architecture such as the Opencores WISHBONE. Ideally, integration of these components would be done in a realible and secure manner. Unfortunately, since many of these cores come from potentially untrusted sources, their use in high-assurance applications becomes extremely limited. This stems from the fact that these cores either come from an untrusted vendor or they have not been evaluated to the same extent as the trused cores. For example, the Mars Rover requires separation between the flight critical and scientific measurement systems simply because the flight critical components require detailed evaluation far beyond that of the measurement ones. A missed bug or vulnerability in the measurement components could affect the flight control components and desecrate the integrity the entire system.

One concern in mix-trusted SoC integration is due to malicious inclusions such as hardware trojans. These trojans can violate security by using hidden circuitry to either covertly transmit information or insert a kill switch into the system. A survey by Tehranipoor et al. [3] covers many of the detection techniques including power and timing-based analyses. The work we present here can help deal with hardware trojans, but requires additional techniques to help mitigate their effect. We can ensure hardware trojans in untrusted cores do not affect trusted ones but we must explicitly assume trusted cores do not have trojans.

Secure mix-trusted integration is not impossible if appropriate techniques are in place to build the system securely from the ground up. By designing a secure computing foundation, information flow can be tightly bounded in the system. Such techniques are hard to come by since information can flow through difficult to detect side-channels[1] in hardware; e.g. the amount *time* a computation takes to execute. Recently, researchers have put effort in developing these strategies, specifically with the use of *information flow tracking* at the lowest digital abstraction: logic gates.

Gate level information flow tracking (GLIFT) [4] uses additional logic to monitor the security level of every bit in the system as they flow through Boolean gates. Similar to information flow tracking at higher abstractions [5], [6], GLIFT associates a single-bit security label (known as taint) to each data-bit and tracks this information

[1] A side-channel is defined as an entity which leaks information but was not intended for communication. The two most common side channels found in hardware are from timing (the data-dependent latency of a computation) and power (the data-dependent power consumed during a computation). In this work we address only logical side-channels (timing), physical ones (power) are out of the scope of this paper.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

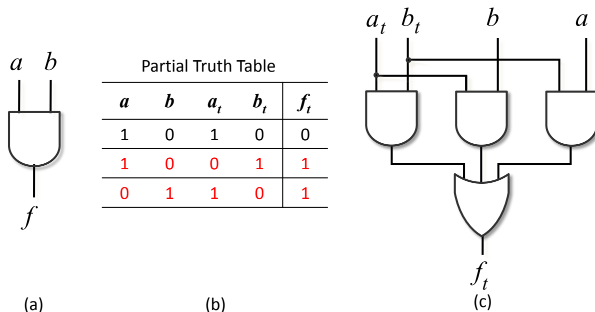
as it flows through the system. This meta-data specifies the security level of every bit in the system and the extra logic gates to presicely monitor this meta-data to determine where the original data is moving. Since GLIFT works at the lowest digital abstraction, it is also capable of tracking information through timing channels as recently demonstrated [7]. This strong property makes it possible for testers and designers to determine whether or not untrusted information is flowing to trusted components so they have a sense of the potential security flaws in their designs. In the past, GLIFT has been used to show how to build a provably secure processor [8] and show isolation in the I2C and USB bus protocols [9]. There has not, to the best of our knowledge, been work showing how GLIFT can be used to show isolation in a larger, realistic SoC which uses components from varying trust.

The goal of this paper is to show how a SoC can be designed using cores from different trust levels and have its security tested using GLIFT. In doing so, we demonstrate that untrusted cores never affect trusted ones. Specifically, we target the WISHBONE [10] SoC protocol using a cross-bar interconnect. We design a realistic system which resembles that of what one might find in high-assurance applications. Specifically, two processors (trusted and untrusted) which wish to share a hardware accelerator (AES encryption unit) in the SoC. Ideally, this sort of behavior should be allowed as long as the untrusted component does not interfere with the trusted one (and thereby compromise the integrity of the system). Using GLIFT, we show how a cross-bar can be designed and tested to be information flow secure such that the untrusted processor never affects the trusted one. This allows the hardware accelerator to be shared in a secure way without causing harmful side effects to the trusted computation. We demonstrate that this isolation is maintained across several different scenarios in which the untrusted processor is attempting to interfere with the trusted one.

II. Gate Level Information Flow Tracking

Gate level information flow tracking (GLIFT) is an information flow tracking technique targeted at the movement of information through Boolean gates. It has been used in a variety of applications, e.g., demonstrating isolation between devices in bus protocols [9] and processes in a microprocessor [8].

As with other information flow tracking methods, GLIFT associates a bit of meta-data (henceforth referred to as taint) and tracks this taint through the system as it executes. This bit of meta-data represents the security of the data (either trusted or untrusted) so that the flow of untrusted information can be precisely monitored. For example, consider a simple AND gate and partial truth table as shown in Figure 1.



a	b	a_t	b_t	f_t
1	0	1	0	0
1	0	0	1	1
0	1	1	0	1

Fig. 1. (a) A simple AND gate. (b) A partial truth table for the tracking logic of an AND gate. $f_t = 1$ *iff* a tainted input affects f. (c) The tracking logic for an AND gate

Here we have a simple AND gate (a) with inputs a and b and also the tracking logic for this AND gate (c) with taint inputs a_t and b_t in addition to the original data inputs. The partial truth table (b) specifies how the GLIFT logic (c) tracks the taints of the intputs to the output. For example, as shown in row 1, if a is tainted ($a_t = 1$ or a is untrusted) with $a = 1$ and b is not tainted with $b = 0$ then no tainted information (from a) flows through the logic gate since the output is always 0 since $b = 0$. In other words, a cannot *affect* the output f of the AND gate in this scenario. The tracking logic captures this property by indicating f as untainted ($f_t = 0$). Conversely, if we consider row 2 in which b is tainted instead of a, then tainted information does flow through the AND gate since b affects the output f. This is captured by the GLIFT logic by labeling f as tainted ($f_t = 1$). Similar truth tables can be constructed for other gate primitives (OR, XOR, etc.) so that GLIFT logic can be created for any gate in the design.

To use GLIFT in practice the existing logic synthesis tools are leveraged to tightly integrate it into the design flow. First, we take a design written in a hardware-description language (HDL) such as Verilog or VHDL at the register-transfer level (RTL). This hardware design is then synthesized to logic gates using Synopsys' Design Compiler and target its `and_or.db` library which contains simple 2-input ANDs, ORs, and inverters. Note that we use this library for the sake of simplicity; more complex libraries can be used as long as the GLIFT logic has been derived *a priori* as previously discussed. Once the logic is in the form of a gate-level netlist, we process this netlist to add the additional GLIFT logic. This process simply takes every gate primitive and replaces it with the appropriate GLIFT logic (this new logic contains both the original and tracking logic).

Once all the pieces are in place, the design can be tested to determine whether or not an information flow exists. This is done by tainting known untrusted regions of the design and simulating execution on input test vectors using a simulation tool such as Mentor Graphics Modelsim. If this tainted information flows to an trusted region, the design has a security vulnerability that the designer must assess. While GLIFT itself does not provide any mechanism for determining *why* there is a security violating information flow, it will always correctly indicate the absence of one. We reserve providing techniques for showing *why* for future work. As of now, it is up to the designer to reason about the flow and put mechanisms in place to eliminate it. GLIFT will, as mentioned, correctly identify the absence of this flow once these mechanisms are added. Note that using GLIFT in this manner will show the absence of a flow for the test vectors used. It does not necessarily guarantee the absence of unintended information flow for all input combinations. Past work has created a solution to this problem called Star-Logic [8], but it is out of the scope of this paper.

To show how this works more concretely, we show how to apply this technique to a realistic SoC with the WISHONE bus architecture. We undergo this test in a similar manner as past work [9]. However, the system we present here is much more realistic and complex; providing a clearer sense as to how GLIFT can be applied to modern designs. The details of this system and how we create an information flow secure interconnect for WISHBONE are presented in the next section.

III. Designing a Secure Crossbar in Wishbone

WISHBONE is a SoC protocol originally developed by the Opencores community. It is a relatively simple protocol that allows easy integration of different cores into a design. WISHBONE itself is very flexible and allows many different interconnect configurations and

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

bus transactions. WISHBONE allows many connectivity configurations including: point-to-point, data-flow, shared bus, and cross-bar interconnect. In this paper, we focus on the cross-bar interconnect since it provides a flexible interface for systems which contain large numbers of cores interacting in parallel.

We wish to demonstrate that multiple cores can access a shared resource in a safe and secure manner. We designed a system which consists of two MIPS-based processors and a 128-bit Advance Encryption Standard (AES) core. The two processors share the AES core over the WISHBONE interface. We assume that one of these processors runs critical code while the other is untrustworthy, e.g., running unknown (potentially malicious) code or not being as thoroughly evaluated as the trusted core. Further details of this system are discussed in the next subsection.

A. Mix-Trusted System with Hardware Accelerator

Our system consists of two MIPS-based processors and a 128-bit AES core. We designed the MIPS based processor and the 128-bit AES was obtained from the Opencores [11] website. All cores are written in Verilog HDL. We chose this configuration because it well suits the common issues found in high-assurance applications. Namely, it is often desirable to share a hardware accelerator in a large SoC with mix-trusted components. Although this system is does not have all the complexity of commercial SoCs it does capture the main idea that multiple mix-trusted cores share common hardware resources and isolation between them should be maintained.

Figure 2 (a) shows the overview of our system. It consists of two of our processors and a 128-bit hardware AES unit. One of these processors is treated as untrusted (U) and the other trusted (T). In other words, we do not trust the behavior of processor U and assume its intentions are to corrupt the execution of T. Our MIPS based processor is fully functional and can execute many of the SPEC 2006 benchmarks (e.g. mcf, specrand, bzip2) [12]. To execute these applications (which are written in C), we used the SESC gcc cross-compiler to compile to MIPS binaries. These binaries are loaded into our processor's memory and the executions are simulated using Mentor Graphics' Modelsim. In order to communicate off-chip, we memory-mapped our processors WISHBONE I/O controller to a region of unused memory space. Since we have a cross-compiler for our processor, we wrote C-applications to push data out of the WISHBONE I/O interface. We wrote different applications for U and T to execute as we discuss later.

We also designed the cross-bar interconnect to handle requests from the processors. The cross-bar interconnect is connected to each processor's WISHBONE controller (Figure 2 (a)). This cross-bar interconnect handles requests from the two processors in a round-robin fashion. This is simply for correctness and to prevent any sort of denial of service. Each processor can perform at most one transaction before having to relinquish control of the bus. It waits for requests from a master and grants access to the slave at the address specified if the slave is available. In our scenario, we have only a single slave: a 128-bit hardware AES unit. Depending on the request type, this AES unit will take the data passed to it (in 32-bit chunks) and encrypt/decrypt a 128-bit block. The processor which requested the bus cycle polls until the transaction is complete and then retrieves the data from the AES unit. Upon completion, the next processor (if it has a pending request) will get access to the AES core.

Note that all the communication between the processor and AES unit are through WISHBONE and its cross-bar interconnect. In this system, since we have both trusted and untrusted processors contending for the use of the AES unit, there is likely to be information flows

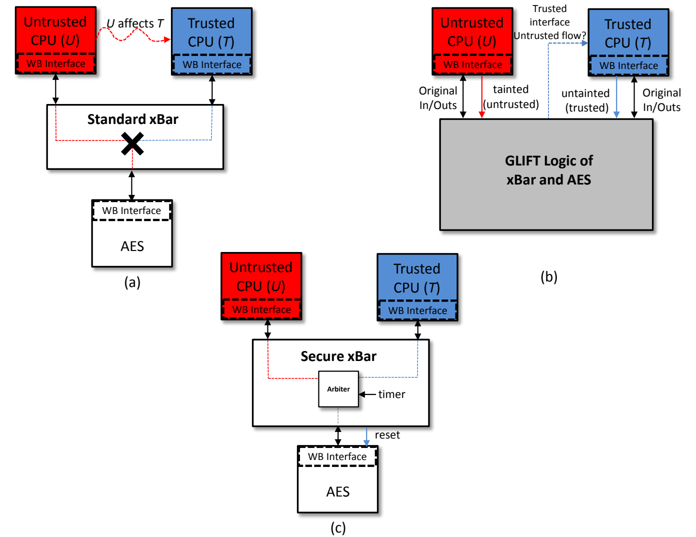


Fig. 2. (a) The system used in our test scenario. This consists of two MIPS-based processors and a 128-bit AES encryption core. U and T contend for the use of the AES core. (b) The system after the AES core, xBar, and interface controllers have their GLIFT logic added. Information is observed to flow from U to T. (c) The final information flow secure system uses a time-multiplexed arbiter with a trusted reset to ensure information flow isolation between U and T. Adding the GLIFT logic to this system shows no information flowing from U to T.

from U to T. Such a flow would violate the integrity of T and should be prevented. Moreover, this interference is not a denial of service attack since it is not possible for U to keep T from completing its work. Still, U can effect when T gets access to the AES block because it must wait for U's transaction to complete. For example, if U never wants to use the bus, and T performs continuous bus transactions, T can finish in some time t. However, if U performs bus transactions every time it is scheduled, T will finish its bus transactions in time $\approx 2t$. Thus U can affect the *time* in which T finishes execution but cannot prevent it from doing so. The next section discusses how we identify information flows in this system and how to eliminate them.

B. Building a Secure Cross-Bar for WISHBONE

To first illict how an information flow occurs from U to T, we test a scenario in which T encrypts a 128-bit block of text using the AES unit and subsequently decrypts the cipher-text to verify the result. In parallel with T, U continuously reads a configuration register on the AES core. We call this program executing on processor U as R_CONF. This scenario was chosen to show an information flow because U is not overwriting any of T's data since it is only reading. In other words, U is not directly corrupting T's data on the AES block and at first glance U seems to be non-interfering with T.

Since we are concerned with the information flow from U to T, we need to look at the information flowing out of U and in to T. To be precise, let $T_{in_t} = \{data_i_t, ack_i_t\}$ be the taint input wires to T from the wishbone logic. We determine whether or not a flow occured by identifying whether any wire in T_{in_t} is every set to 1. To do so, we must track the flow of information through the cross-bar, the AES unit's WISHBONE controller, and the AES unit itself. To track this flow of information, we follow the same method presented in Section II. Namely, we process the cross-bar and the AES unit with its WISHBONE interface through synthesis using Synopsys' Design Compiler to achieve a gate-level netlist. Subsequently, we add the GLIFT logic to these components and re-insert this logic into the

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

p on U	τ on T	Description	Flow in Secure xBar	Flow in Base xBar
AES	MM	U encrypts, decrypts, and validates result; T executes matrix-multiply	NO	NO
MM	AES	MM executes on U while T encrypts, decrypts, and validates result	NO	YES
R_CONF	AES	Repeatedly read the status register on AES core	NO	YES
R_ALL	AES	Read entire address space of AES core	NO	YES
W_ALL	AES	Write entire address space of AES core	NO	YES
AES	AES	Both encrypt, decrypt, and verify result	NO	YES

TABLE I
DESCRIPTION AND RESULTS OF DIFFERENT APPLICATIONS EXECUTED ON U AND T. UNTRUSTED FLOWS ARE IDENTIFIED IN THE BASE CROSS BAR FOR MOST SCENARIOS AND NONE ARE IDENTIFIED IN THE SECURE CROSS-BAR. FLOWS DO NOT OCCUR IF T DOES NOT USE THE WISHBONE INTERFACE AS IN THE CASES OF RUNNING MM.

system as shown in Figure 2 (b). We then execute R_CONF on U by simulating the Verilog in Modelsim. From the simulation, as shown in Figure 3, a tainted flow is observed entering T's inputs as soon as it requests an AES transaction ($\{data_i_t, ack_i_t\} = \{0xF \cdots F, 1\}$). Since we only tainted the ouputs of U, it must be the source of this tainted information flow.

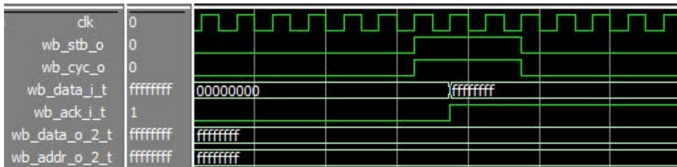


Fig. 3. Waveform showing tainted information flow. As soon as T requests access to the AES unit (wb_stb_o = wb_cyc_o = 1) tainted information flows to its inputs ({data_i_t, ack_i_t} = {0xF⋯F, 1}). U's outputs were the only marked as tainted, so this flow must have originated from U.

This flow occurs because U and T contend for the use of the encryption unit. Specifically, U affects the execution of T indirectly by its use of the AES unit. This flow can be regarded as occuring through a timing channel. That is, U is able to affect the *time* in which T finishes its computation (U is only reading and therefore does not directly affect the computation of T). Such channels can violate the integrity of the design because they can potentially violate real-time constraints where T must meet a critical deadline but is unable to because of U. To solve this problem, we put in place a way for U to never affect T's use of this resource. Specifically, we introduce a time-multiplexed arbiter with a *trusted reset* to the cross-bar which forces T and U to operate in mutually exclusive time slots as shown in Figure 2 (c). Upon expiration of a time-slot, the logic is restored to a known state to ensure harmful content is left behind. As we see in the next section, this new cross-bar eliminates this untrusted flow.

C. Secure Cross-Bar Evaluation

To demonstrate the lack of information flow using this new cross-bar, we construct several different programs which have malicious characteristics of causing interference to the trusted computation on T. Specifically, we show non-interference for a fixed set of programs. Non-interference states that U should never affect T through any sort of digital information. This includes both directly corrupting the data of T or affecting the time in which programs on T take to complete. This ensures not only the integrity of the data on T, but also the integrity of the timing of the computation. To demonstrate this property for a set of programs, let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of programs to be run on U. We want to show non-interference with respect to P by demonstrating that no untrusted information flows to the inputs of T:

$$\forall p \in P.\ S(p \,||\, \tau) \overset{c}{\Rightarrow} T_{in_t} = \{0, 0\} \tag{1}$$

where $S(p \,||\, \tau)$ is the system executing with p on U and τ on T and $\overset{c}{\Rightarrow}$ is an implication over all clock cycles c. T_{in_t} is the set of taint inputs from the wishbone cross-bar as previously defined. This definition says that for any program p in a set P, when executing p on U with some trusted computation on τ on T, no untrusted information from U flows to the inputs of T during any clock cycle. Since GLIFT can also capture information flowing through timing channels as mentioned in the previous section, this includes information which affects the time in which τ takes to complete.

For our particular test scenario, we build the set $P = \{MM, R_CONF, R_ALL, W_ALL, AES\}$. MM is a simple matrix multiply program. R_CONF is the same program as before which continuously reads a configuration register on the AES core. R_ALL attempts to read the entire address space associated with the AES core. W_ALL attempts to write the entire address space associated with the AES core. Lastly, AES uses the AES core to encrypt then decrypt some information. All of these applications are written in C, compiled to MIPS, and loaded on to their respective processor's instruction memory. Table I presents an interesting subset of our test cases and summarizes the outcomes. We do not present all results due to space constraints but observed that Definition 1 holds for each τ we tested.

For all cases in which τ accesses the WISHBONE fabric, untrusted information flows from U to T in the unsecure cross-bar, thus violating Definition 1. One interesting case is when MM and AES are run on T and U respectively. In this case, no untrusted information flows to T simply because τ never accesses the AES core. Its execution is independent of the behavior of U. Conversely, another interesting case arises when U runs MM and T runs AES. In this case, even though p is not using the AES core, the *lack* of its use still affects the behavior of τ. This lack of use allows τ to finish faster than if p were accessing it; a flow of information. GLIFT indicates no flow ($T_{in_t} = \{0, 0\}$) for all applications when the secure cross-bar is used. In other words, non-interference is upheld for these computations on U.

It is important to make a couple of notes on this solution. First, the arbiter only time-multiplexes this specific resource and not the cross-bar as a whole. The goal of the cross-bar interconnect is to allow parallelism; multiplexing the entire cross-bar eliminates this flexibility. This parallelism can still be maintained since U can be granted access to other devices in the system in parallel with T and isolation can still be maintained. In addition, ideally this property (Definition 1) would be shown for all possible programs on U to demonstrate complete non-interference. However, such an exhaustive test would be impractical in this case. Some recent work on GLIFT

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

has made an effort to solve this problem by introducing Star-Logic [8] which uses an abstract execution to make exhaustive testing possible. Unfortunately most of this work is still in its early stages, but we plan to employ these techniques in future research.

IV. Conclusion

Computers are finding themselves at the heart of avionics, medical devices, military applications, automobiles, and many other critical aspects of our lives. Building these systems in a secure manner requires strict design practices and tools. In this paper, we showed how mix-trusted IP cores can be integrated in a secure manner. By using gate-level information flow tracking to show information flow isolation between trusted and untrusted cores, we have constructed a secure cross-bar interconnect for the WISHBONE SoC bus architecture. This powerful property makes it possible to integrate mix-trusted cores and verify the security of their interactions. This ultimately reduces the cost and time associated with development and makes using untrusted cores in high-assurance applications more of a possibility.

References

[1] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy*, pages 129 –142, 2008.

[2] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *Proceedings of IEEE Symposium on Security and Privacy (“Oakland”) 2010*, pages 447–462, 2010.

[3] M. Tehranipoor and F. Koushanfar. A survey of hardware trojan taxonomy and detection. In *IEEE Design and Test*, 2010 2010.

[4] Mohit Tiwari, Hassan Wassen, Bita Mazloom, Shashidhar Mysore, Frederic Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of ASPLOS 2009*, 2009.

[5] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS 2004*, pages 85–96, 2004.

[6] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO 2004*, pages 221–232, 2004.

[7] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. A practical testing framework for isolating hardware timing channels. In *Design Automation and Test in Europe (DATE)*, 2013.

[8] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan Valamehr, Timothy E. Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of ISCA 2011*, pages 189–200, 2011.

[9] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Proceedings of Design Automation Conference (DAC) 2011*, pages 254 –259, 2011.

[10] Wishbone specification. http://opencores.org/opencores ,wishbone.

[11] Opencores.org. 128-bit verilog aes core. http://opencores.org/project ,systemcaes, April 2010.

[12] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, pages 1–17, 2006.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.