Secure and Flexible Trace-Based Debugging of Systems-on-Chip

JERRY BACKER, New York University
DAVID HELY, University of Grenoble Alpes
RAMESH KARRI, New York University

31

This work tackles the conflict between enforcing security of a system-on-chip (SoC) and providing observability during trace-based debugging. On one hand, security objectives require that assets remain confidential at different stages of the SoC life cycle. On the other hand, the trace-based debug infrastructure exposes values of internal signals that can leak the assets to untrusted third parties. We propose a secure trace-based debug infrastructure to resolve this conflict. The secure infrastructure tags each asset to identify its owner (to whom it can be exposed during debug) and nonintrusively enforces the confidentiality of the assets during runtime debug. We implement a prototype of the enhanced infrastructure on an FPGA to validate its functional correctness. ASIC estimations show that our approach incurs practical area and power costs.

CCS Concepts: • **Security and privacy** → *Embedded systems security*;

Additional Key Words and Phrases: System-on-chip, debug traces, secure debug

ACM Reference Format:

Jerry Backer, David Hely, and Ramesh Karri. 2016. Secure and flexible trace-based debugging of systems-on-chip. *ACM Trans. Des. Autom. Electron. Syst.* 22, 2, Article 31 (December 2016), 25 pages.
DOI: <http://dx.doi.org/10.1145/2994601>

1. INTRODUCTION

Hardware debug instruments are circuits added to the design of a system-on-chip (SoC) to facilitate post-silicon debugging. These instruments allow one to capture real-time performance statistics, observe and modify values of internal signals and registers, and examine the disassembly of instructions executed [Vermeulen and Goossens 2014; Orme 2008]. Examples of debug instruments used in commodity SoCs are internal and boundary scan chains, hardware performance counters, defeaturing bits, microcode patches, and real-time signal tracing. The ability of debug instruments to observe the inner workings of the SoC can be used as a backdoor for attacks. For example, studies in Yang et al. [2004] and Chiu and Li [2012] show how an adversary can use the internal and boundary scan chains to leak SoC cryptographic keys. In another example, Uhsadel et al. [2008] show how one can use hardware performance counters to leak sensitive instructions and data. In this work, we explore the threat of an adversary using signal tracing to leak sensitive data. This attack is more practical because it requires no side-channel analysis and can be initiated with minimal knowledge of the SoC design.

1.1. Motivation

Signal tracing allows debuggers to trace values of internal signals of the SoC in real time and with negligible execution time overhead. Signals chosen for tracing

Authors' addresses: J. Backer and R. Karri, Tandon School of Engineering, New York University, Brooklyn, NY 11201; emails: {jerry.backer, rkarri}@nyu.edu; D. Hely, LCIS, University of Grenoble Alpes.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1084-4309/2016/12-ART31 \$15.00

DOI: <http://dx.doi.org/10.1145/2994601>

are relevant to instructions executed, memory transfers and requests, and the SoC operating state [Vermeulen and Goossens 2014; Intel 2015; ARM 2011]. The values of traced signals are analyzed offline to detect hardware and software bugs and to explore new SoC designs. Several third-parties involved in the design and maintenance of the SoC (e.g., software vendors, distributors, equipment manufacturers) use trace-based debugging at different stages of the SoC life cycle [Orme 2008].

Since signal tracing exposes values of internal signals, it may trace SoC assets that have confidentiality requirements. For example, cryptographic keys used for digital rights management (DRM) can be exposed via traces of memory transfers. A rogue debugger can leverage this to illegally leak the keys and use them to bypass DRM protection [Coburn et al. 2005]. In another example, instructions of a proprietary firmware (e.g., modem signal processing) can be exposed via traces of instructions executed. A rogue debugger can leak such firmware to reverse engineer it [Goodspeed and Francillon 2009] or to use its proprietary content (e.g., algorithm, data).

1.2. Related Work on Secure Debug

To balance security of assets and observability of a debug infrastructure, Ray et al. [2015] establish five requirements: (1) *high-volume manufacturing* (HVM), which highlights the need for correct and consistent debugging results regardless of the security features; (2) *reusability*, which specifies the need for security features to enforce all access use cases of an asset and to be adaptable to different SoCs; (3) *late variability*, which specifies the need for security features to be adaptable to late changes in the design of the debug infrastructure; (4) *self-security*, which specifies that the security features should not expose new threats to the SoC, and; (5) *architecture*, which specifies the need for a centralized security design to facilitate its verification. We use these requirements to evaluate existing approaches that could be used to secure debug traces.

One can permanently disable the debug infrastructure after silicon validation (i.e., by blowing fuses). However, this approach violates the late variability and reusability requirements because trace-based debugging is needed throughout the SoC life cycle [Orme 2008]. One can use *dummy* assets during trace-based debugging. This is similar in spirit to work done in Lee et al. [2006], where dummy flip-flops are added to prevent scan-based attacks on cryptographic chips. Yet this approach does not work for software verification and optimization because functional correctness of execution is needed. This approach thus violates the reusability requirement. In addition, the execution path with dummy assets may differ from the correct execution, violating the HVM consideration requirement.

One can consider a secure test port approach such as that of Rosenfeld and Karri [2010], Pierce and Tragoudas [2013], or Dworak et al. [2013]. The basic idea here is to lock the debug access port (e.g., JTAG) and to only allow trusted debuggers to access the debug infrastructure after successful authentication. This approach provides either complete or no access to debugging. It cannot secure all use case scenarios of assets and thus does not meet the reusability requirement. One can use a manual blacklisting approach, where traceable signals that expose assets are analyzed and are either unlocked or locked for debug. However, as highlighted in Ray et al. [2015], this approach does not meet the reusability requirement, as it is tedious and error prone.

In ARM-based SoCs, the debug infrastructure provides two debug modes: secure and nonsecure [ARM 2013]. If authenticated in secure mode, the debugger can trace instructions of secure and privileged software (potential assets in our case). Otherwise, only user-level software can be traced. This approach does not meet the reusability requirement, as it gives either complete access to all secure software or no access to any secure software. For example, consider an untrusted debugger that owns a

privileged or secure code as an asset. If authenticated in secure mode, this debugger can trace its own secure software as well as all other secure software. If authenticated in nonsecure mode, this debugger cannot analyze its own code.

The patent in Hardy et al. [2010] presents a software authentication approach for secure debug. On debug boot-up, a signature of the software is generated using a cryptographic key stored in SoC nonvolatile memory. This signature is then compared to a golden debug signature. If the comparison matches, the boot process completes, the debug instruments are enabled, and the software is allowed to execute. Otherwise, booting is halted, debug is disabled, and the software is not executed. This approach primarily focuses on the integrity of the software (instead of access rights of the debugger) and can be used in concert with other secure debug features.

1.3. Contributions

As detailed in Section 1.2, existing secure debug solutions do not meet the reusability requirement established in Ray et al. [2015]. This is a critical requirement because it emphasizes the need for a secure debug infrastructure that protects assets under all use scenarios while being adaptable to multiple SoC products. We extend the SoC trace-based debug infrastructure to meet the reusability requirement as well as the other requirements of secure debug. We propose an automated runtime approach to filter which assets to trace based on their debug confidentiality policy. This automated filtering method covers all use cases of an asset, overcoming the limitations of existing methods. Our contributions are the following:

- (1) We explore the feasibility of maliciously leaking data and firmware assets via debug traces, provide case examples for this attack, and discuss its limitations.
- (2) We design a flexible secure debug infrastructure to dynamically monitor assets at runtime and enforce their access policies accordingly. Our approach ensures that all trace instances of a given asset are first filtered before they are traced.
- (3) We implement a prototype of the secure debug infrastructure on a Xilinx Spartan-3 FPGA and estimate its area, power, and performance costs on an ASIC.

The article is organized as follows. In Section 2, we detail a baseline trace-based debug infrastructure using ARM CoreSight as reference and review SoC assets that we consider in this work. In Section 3, we detail the threat model and illustrate case examples of the attack. We present the secure debug infrastructure in Section 4 and detail our implementation in Section 5. In Section 6, we discuss limitations of our implementation and how they can be overcome. We conclude the article in Section 7.

2. BACKGROUND

2.1. Trace-Based Debugging

Trace-based debugging allows SoC integrators to observe values of internal signals of IP cores embedded in the SoC. The main difference between trace-based debugging and other post-silicon debug techniques (e.g., scan-based debug) is the ability to capture the values of the signals in real time without the need to stop execution. This provides the benefits of faster debugging, timestamped logging of traces, and cycle-accurate and deterministic debugging. With these benefits, trace-based debugging is used for SoC architectural exploration, software verification and optimization, and in-field maintenance [Orme 2008]. In addition, third parties involved in the design and maintenance of the SoC have adopted trace-based debugging for their verification purposes. Examples of such third parties are outside equipment manufacturers (OEMs), outsourced semiconductor and test (OSAT) companies, and software and middleware developers.

Fig. 1. Trace-based debugging.

Figure 1(a) illustrates the circuitry added to enable trace-based debugging of an IP. Figure 1(b) shows the trace-based debug infrastructure of the reference SoC used in this study. The figures are illustrative of the ARM CoreSight [Orme 2008]. We choose ARM CoreSight for illustration because it is used in commodity SoCs. For a given IP core, the integrator chooses specific internal signals to expose for tracing. The signals are chosen based on the IP type. For an IP that executes instructions (e.g., CPU, DSP), signals are chosen for instruction or data tracing. Instruction tracing signals can be address and disassembly of instructions executed [Gaisler et al. 2015a], target addresses of branch instructions, and context ID of the software [ARM 2011]. Data tracing signals can be addresses and data of read and write memory transfers and requests [ARM 2011]. For an interconnect IP (e.g., system fabric), memory transfer signals such as data, address, and transfer attributes (e.g., direction, privilege, size) can be chosen for tracing [ARM 2008; Gaisler et al. 2015a]. For an accelerator IP (e.g., cryptographic IP), signals are chosen ad hoc based on their importance to the functionality of that IP [Liu and Xu 2009]. Signals chosen for tracing are connected to a signal filter. The filter can be configured via the SoC JTAG port to select which signals to trace when the IP is under debug (CUD) and under what executing conditions (e.g., context ID, memory address range) to trace them. The filter uses a trace generator to compress and packet the values of the traces, then funnels them out to a debug port or a memory buffer.

2.2. SoC Assets

An SoC typically includes sensitive data, software, and firmware that have specific security requirements. This is because such data and firmware, commonly referred to as *assets*, play a critical role in the functionality of the SoC, have proprietary information with financial value, or are part of the SoC security apparatus. Examples of assets found in commodity SoCs are cryptographic keys, configuration and calibration data, firmware signatures or certificates, unique device IDs, and proprietary firmware. Assets are stored in nonvolatile memory and can be provisioned by the SoC integrator, a third-party IP (3PIP) vendor, the OEM, or a middleware/software vendor. Throughout the rest of the article, we refer to such a provisioning party as an *asset owner*. Note that an asset owner can also be a debugger (e.g., OEM, SoC integrator, software vendor).

An asset can have a confidentiality, integrity, or availability requirement. For example, for confidentiality, a cryptographic key used during device authentication must be visible only to the asset owner to avoid malicious authentication. For integrity, an untrusted party must not modify configuration data to avoid compromising the SoC operating state. For availability, calibration data of the SoC must always be available

Table I. Examples of Data and Firmware Assets with Confidentiality Requirements

	Examples	Impact of Leaking Asset
Data	OEM DRM key	Access to protected content [Coburn et al. 2005]
	SoC integrator device ID	Authentication bypass [Ayday and Rajagopal 2011]
	SoC integrator HDCP key	Access to protected content [Johnson et al. 2011]
	OEM secure boot key	Loading of arbitrary firmware [Zhang et al. 2007]
Firm.	OEM boot loader	Gaining root access [Goodspeed and Francillon 2009]
	Proprietary middleware	Leak of proprietary algorithms

to avoid permanent denial of service. This work focuses on assets with confidentiality requirements. Such assets can be cryptographic keys, unique device IDs used in authentication, and proprietary firmware. Table I lists examples of such assets in commodity SoCs and the security impacts of exposing them to untrusted parties. End-user data such as contact lists and personal media files are not considered.

3. THREAT MODEL

We assume that the SoC is designed using a horizontal model: the integrator procures 3PIP cores and integrates them according to specifications of the OEM. The design is then sent to an outside foundry where the SoCs are manufactured; the SoCs are tested, assembled, and distributed by an OSAT company and an outside distributor. Third-party OS and middleware developers provide the majority of the SoC software layer. We assume that the integrator is the only trusted party in the SoC life cycle. In this respect, the integrator sets up the trace-based debug infrastructure and incorporates the security features proposed in this work. We also assume that the integrator has in place a security mechanism such as that of Khosravi et al. [2014] to ensure the confidentiality of assets during provisioning and implements a mechanism such as that of Coburn et al. [2005] to ensure that only privileged software can access the assets during functional execution.

The adversary is a rogue debugger that uses debug traces to illegally leak an asset. This debugger can initiate such an attack at any stage of the SoC life cycle. For example, during post-silicon verification, the OSAT can try to leak proprietary firmware of the OEM. In another example, during in-field maintenance, a middleware vendor can try to leak the DRM key of the OEM. Attacks that use other debug instruments, such as scan chains, microcode patches, and defeaturing bits, are outside the scope of this work.

3.1. Illegal Leaking of SoC Assets via Debug Traces

The attack works in three steps: (1) *signal tracing*, where the adversary configures signal filters to generate traces relevant to the asset; (2) *trace decompression*, where the adversary decompresses the traces to reconstruct the SoC execution flow; and (3) *asset extraction*, where the adversary obtains the target asset by either combining traces or by reading the asset from the decompressed traces. The inner workings of each step depend on the debug infrastructure. For example, in the LEON3 debug infrastructure [Gaisler et al. 2015a], traces are not compressed and the second step is not needed. In ARM CoreSight, traces are compressed and the adversary can decompress them using tools provided by ARM. The adversary can also design tools to automate decompression because compression in ARM CoreSight primarily involves omitting leading zeros as well as bits that are repeated in successive traces [ARM 2011].

3.1.1. Attack Example: Leaking DRM Device Key. The DRM device key is a cryptographic key used for DRM protection [ARM 2009]. The key decrypts the rights object of

DRM-protected content. The rights object stores the content key to decrypt the protected content. If the adversary has access to the device key, the content keys can be obtained and the adversary can decrypt the protected files. To leak the key, the adversary first assumes its size. This can be based on public documentation that details DRM implementation practices or on common cryptographic key sizes. For illustration, we assume a 256-bit key. As a device key, it is stored in nonvolatile memory:

- (1) *Signal tracing*: The adversary can use traces of the system fabric, CPU, or cryptographic accelerator to leak the key. We consider the case of using the system fabric traces. The adversary configures the signal filter of the fabric to trace data transfers for only 256-bit values. The adversary then runs the multimedia application that uses the DRM-protected content until the device key is assumed to have been used. For example, the adversary can stop debugging after DRM-protected content has been downloaded and begins to play. This is because a successful playback indicates that the device key has decrypted the rights object.
- (2) *Trace decompression*: This step is straightforward, as the adversary can build decompression tools or reuse existing tools to decompress the traces of the fabric.
- (3) *Asset extraction*: To determine which 256-bit data is likely to be the DRM device key, the adversary considers how the key may have been used. For example, the key is likely first read from its nonvolatile memory, then written to the key register of the cryptographic accelerator to decrypt the rights object. In such case, the adversary can search the decompressed traces for all read/write transfer pairs of the same data. The adversary can then test the data of each pair to determine which one is the key. For each test, the adversary tries to decrypt the rights object of a DRM-protected file to obtain the content key. Then, the adversary uses the content key to try to decrypt the DRM content. If the decrypted content is successfully played, then the DRM device key is the data of that read/write transfer pair.

3.1.2. Attack Example: Leaking Proprietary Firmware. Instruction tracing alone (e.g., via the CPU) may not be sufficient to obtain the complete execution flow of a proprietary firmware. For example, ARM CoreSight does not expose the disassembly of executed instructions. Instead, it only traces the target addresses of committed indirect and direct branches [ARM 2011]. We detail an attack to overcome this issue.

To leak a proprietary firmware, the adversary must run the firmware twice to obtain a different set of traces for each run. In one run, the adversary configures the signal filter of the CPU to obtain traces for the target addresses of branch instructions. In the other run, the adversary configures the signal filter of the system fabric to obtain traces of addresses and values of instruction fetches. The adversary then combines the two traces to get the firmware execution flow and its disassembly. To evaluate the practicality of the attack, we augment the LEON3 SPARC CPU from Gaisler et al. [2015b]. To mimic the tracing of branch targets, we modify the LEON3 CPU to output the target addresses of branch instructions. To mimic the tracing of the system fabric, we modify the memory controller of the CPU to output the address and value of each memorytransfer (instruction and data). We implement the modified CPU on a Xilinx Spartan-3 FPGA and use the basicmath workload from the MiBench suite [Guthaus et al. 2001] in our experimental attack. Figures 2, 3, and 4 show snapshots of the stages of the attack on the modified LEON3 CPU. The attack works as follows:

- (1) *Signal tracing*: The adversary configures the signal filter of the CPU to generate traces for target addresses of all branches. The adversary runs the firmware until completion or after a significant portion of its code has been performed and collects the traces. Then, the adversary configures the system fabric filter to get traces

Fig. 2. Snapshot of CFG extraction.

Address	Instruction	Instruction Disassembly
0x40000000	0x88100000	0x40000000 mov %g0, %g4
0x40000004	0x09100068	0x40000004 sethi %hi(0x4001a000), %g4
0x40000008	0x81C12348	0x40000008 jmp %g4 + 0x348
0x4000000C	0x01000000	0x4000000C nop
0x4001A348	0x83480000	0x4001A348 rd %psr, %g1
0x4001A34C	0x83306018	0x4001A34C srl %g1, 0x18, %g1
0x4001A350	0x82086003	0x4001A350 and %g1, 3, %g1
0x4001A354	0x80A06003	0x4001A354 cmp %g1, 3
0x4001A358	0x12800006	0x4001A358 bne 0x4001A370
0x4001A35C	0x01000000	0x4001A35C nop
0x4001A370	0x88100000	0x4001A370 mov %g0, %g4
0x4001A374	0x09100068	0x4001A374 sethi %hi(0x4001a000), %g4
0x4001A378	0x81C12224	0x4001A378 jmp %g4 + 0x224
0x4001A37C	0x01000000	0x4001A37C nop
0x4001A360	0x83444000	0x4001A360 rd %asr17, %g1
0x4001A364	0x05000008	0x4001A364 sethi %hi(0x2000), %g2

(a) Decompressed fetches

(b) Disassembled instruction fetches

Fig. 3. Snapshot of instruction disassembly.

BB ₁	BB ₂
0x40000000 mov %g0, %g4	0x4001A348 rd %psr, %g1
0x40000004 sethi %hi(0x4001a000), %g4	0x4001A34C srl %g1, 0x18, %g1
0x40000008 jmp %g4 + 0x348	0x4001A350 and %g1, 3, %g1
0x4000000C nop	0x4001A354 cmp %g1, 3
	0x4001A358 bne 0x4001A370
	0x4001A35C nop
	0x4001A360 rd %asr17, %g1
	0x4001A364 sethi %hi(0x2000), %g2

Fig. 4. Disassembly of BB₁ and BB₂.

for addresses and values of all instruction fetches. The adversary then runs the firmware up to the same point as before and collects the traces.

- (2) *Trace decompression*: The adversary uses existing tools to decompress the traces of the branch target addresses and those of the fetched. Figures 2(a) and 3(a) show some of the decompressed traces for branch target addresses and instruction fetches for our experimental attack.
- (3) *Asset extraction*: This stage has three parts: control flow graph (CFG) extraction, instruction disassembly, and firmware extraction:
 - (a) **CFG extraction**: The adversary assumes that the branch target addresses are traced in the order they are executed since they are generated in real time. The adversary assigns a basic block ID for each unique branch target. For example, for our experimental attack, Figure 2(a) shows that the first five unique target address traces of the basimath benchmark are 0x40000000, 0x4001A348, 0x4001A224, 0x4001A2D8, and 0x4001A308. The adversary thus assigns the basic block IDs BB₁, BB₂, BB₃, BB₄, and BB₅ to the respective unique branch target addresses. Using the order in which the target addresses were traced, the adversary forms the runtime CFG of the software at the basic block granularity. Based on the order of the traces in Figure 2(a), the adversary forms the runtime CFG in Figure 2(b). Note that BB₅ has two iterations as it was traced consecutively in Figure 2(a).
 - (b) **Instruction disassembly**: The adversary disassembles the instruction of each address/instruction pair. For example, for the first 32-bit SPARC instruction 0x88100000 in Figure 3(a), the adversary disassembles it into `mov %g0, %g4`, as shown in Figure 3(b). Since this process can be long and tedious, the adversary can use an automated disassembler tool such as the one in ODA [2016]. We emphasize that Figure 3(b) does not represent the order in which the instructions are committed but instead the order in which they are fetched.
 - (c) **Firmware extraction**: The adversary uses the disassembled instructions to obtain the instructions of each unique basic block. For a given basic block, the adversary finds its first instruction from the disassembled instructions. Using our example, the first instruction of BB₁ is at address 0x40000000 and its disassembly in Figure 3(b) is `mov %g0, %g4`. The adversary then calculates the address of the next instruction and searches for its disassembly. Using BB₁, the next instruction is at address 0x40000000 + 4 because SPARC instructions are four bytes. From Figure 3(b), we see that the instruction at this address is `sethi %hi(0x4001a000), %g4`. The adversary continues this process until a branch instruction is observed. If the branch is conditional, the adversary calculates the target address and compares it to the address of the next basic block in the runtime CFG. If the comparison matches, the adversary assumes that the branch was taken and completes the disassembly of the current basic block. Otherwise, the adversary assumes that the branch was not taken and goes to the next instruction of the basic block. In case of an unconditional branch, the adversary completes the disassembly of the basic block and goes to the next executed one based on the runtime CFG. Using our example, the branch instruction of BB₁ is observed at address 0x40000008 and is an unconditional branch in Figure 3(b). The disassembly of BB₁ thus completes and the adversary begins the next basic block. Note that SPARC architecture uses a branch delay slot and that instruction (at address 0x4000000C in our case) is added to the basic block. Figure 4 shows the disassembly of BB₁ and part of BB₂. Once all basic blocks are disassembled, the adversary arranges them in the order of the runtime CFG. This forms the execution flow of the target firmware and allows

the adversary to understand its inner workings and determine sensitive assets such as proprietary algorithms.

The CPU cache hierarchy has no impact on the attack. This is because if an instruction misses in the cache, it is fetched from memory via the system fabric and its disassembly is traced. If an instruction never misses in the cache, this is because it was brought as part of a cache line to handle the cache miss of another instruction. Therefore, this cache line was fetched via the system fabric, and all instructions in that cache line can be disassembled. The adversary can also detect mispredicted instructions and omit them from the disassembly of the basic blocks. This is because the adversary relies on the committed target addresses of branch instructions to determine when to branch. For example, in Figure 4, BB_2 has a conditional branch instruction at address $0x4001A358$. The branch target address is $0x4001A370$. The adversary checks the runtime CFG in Figure 3(a) and sees that the next basic block BB_3 has a different address than the target of the conditional branch. The adversary thus concludes that the branch was not taken and searches for the next instruction of BB_2 at address $0x4001A360$.

3.1.3. Limitations of Attacks. The attacks can be limited during memory-intensive periods. This is because, during such periods, the buffer of the signal filter for the system fabric may overflow, leading to omitted traces. For example, consider the attack aimed at leaking proprietary firmware. If the attack is initiated while several other IP cores are simultaneously accessing memory via the system fabric, or if the target firmware is CPU intensive, some instruction fetch traces may be omitted and the adversary may not be able to find all instructions in some basic blocks. The attack on the proprietary firmware is not feasible if the firmware is executed from a local memory. For example, the CPU may have a local ROM that stores the proprietary firmware. All instruction fetches go through the local ROM, and there is no interaction with the system fabric. Therefore, the adversary cannot trace the instruction fetches and cannot perform the attack. The attack can also be limited if the branch target addresses are traced out of order; this is because the adversary may not be able to form the correct runtime CFG of the program, which is needed for successive steps of the attack.

4. SECURE TRACE-BASED DEBUG INFRASTRUCTURE

We present a secure debug infrastructure to protect the confidentiality of memory-mapped assets during trace-based debugging. The infrastructure (1) maintains flexibility of debugging (e.g., the infrastructure does not disable debugging in case of an attack but instead silently obfuscates the traces relevant to the assets and continue the debugging process), (2) requires minimal additions to the SoC design flow, and (3) minimizes changes to IP cores to reduce design effort and to be scalable for low-cost SoCs. The proposed secure infrastructure has three components:

- Secure asset tagging:** The SoC integrator tags each asset with an ID of the asset owner. The ID itself has no confidentiality requirements; it is simply used to indicate to which debugger(s) the asset can be exposed via debug traces.
- Debugger authentication:** By default, the JTAG instruction register is locked and a debugger has no access to the debug infrastructure. To initiate debugging, a debugger must first be authenticated, and its tag ID is returned if successful.
- Asset filtering:** During debug, if an asset is being traced, its tag is compared to the ID of the authenticated debugger. If the comparison matches, the asset is traced; otherwise, it is obfuscated.

The debugger authentication component is similar to secure JTAG methods in Rosenfeld and Karri [2010] and Dworak et al. [2013]. The main challenges of our approach are to tag data and firmware assets uniformly, monitor where the assets are located during runtime debug, and propagate their tags to the filters accordingly.

4.1. Secure Asset Tagging

One way to tag the assets is to expand the size of the memory where they are stored. For example, using 4-bit tags, a 256-bit cryptographic key would need 260 bits of storage. This way, the tag can be trivially propagated when the key is accessed. However, this requires changes to the IP cores to update the size of internal buses. Moreover, this approach is not practical for firmware assets.

We propose to tag assets according to their base addresses in nonvolatile memory, and to store the tags in a read-only lookup table (LUT). We refer to this LUT as the tag LUT. This approach requires no changes to the internal logic of the IP cores and can tag data and firmware assets uniformly. At design time, each asset owner gives the SoC integrator the base address and size of each of its assets. The integrator assigns a tag ID to an asset owner and tags its assets using that ID. The length of the tags is equal to the total number of asset owners, and each bit in a tag represents an owner. Therefore, at most one bit in a tag ID is set to 1. For each asset, the SoC integrator stores its base address, size mask, and tag ID in an entry of the tag LUT.

We illustrate the asset tagging process and the interaction of the stakeholders (e.g., integrator, asset owners) in the following example. At design time, the SoC integrator determines that a total of four asset owners will provision assets for the SoC. This number may be adjusted at any time between design and mass production, as more asset owners (e.g., middleware vendors) get involved. One such asset owner is the OEM and is represented by the most significant bit of the tag. Therefore, the OEM tag ID is $b'1000$. The OEM has a 32KB proprietary firmware at base address $0x00000000$. The integrator stores the base address $0x00000000$, the size mask $0x00007FFF$, and the tag $b'1000$ in an entry of the tag LUT.

There may be scenarios where multiple debuggers need permission to a given asset. To compute the tag of such an asset, we perform a logic OR of the tag ID of all debuggers that need permission. For example, consider that debuggers with tag IDs $b'1000$ and $b'0001$ need access to a given asset. The tag for this asset is assigned as $b'1001$.

4.2. Debugger Authentication

At design time, the integrator generates a set of cryptographic keys $K0$. Each key $K0_i \cup K0$ is assigned to a debugger that is also an asset owner and is associated with the tag ID of that owner. The integrator also generates a key KP that is provided to the other debuggers. These debuggers are given the same tag ID where all bits are set to 0 to indicate that they are prohibited from observing assets via debug traces.

Figure 5 illustrates the debugger authentication protocol. The debugger uses a workstation to connect to the SoC under debug. Generally, the workstation has a direct connection to the SoC. However, in some cases, a network may interface between the two parties. For example, Lau and Varshney [2014] describe a test environment with an OS-independent debug software. This is accomplished by hosting the debug software and JTAG drivers on a test server and by allowing the debugger to access the debug software via a Web browser. Under such a test environment, we must ensure that the network cannot be leveraged to bypass the authentication protocol.

The SoC integrator adds an authentication module for the protocol; the keys $K0$ and KP are stored in the module. To initiate debugging, the debugger sends an UNLOCK request along with its tag ID to the module. Upon receiving the UNLOCK request, the module generates a challenge C_p . The module uses the debugger tag ID to retrieve the

Fig. 5. Debugger authentication ©IEEE 2015.

associated key K_M , where $K_M \in (K_P \cup K_0)$. The module calculates a keyed-hash message authentication code (HMAC) of the challenge C_D : $HMAC(C_D, K_M)$. Simultaneously, the module forwards the challenge C_D to the debugger via the JTAG port. Upon receiving the challenge, the debugger uses its cryptographic key K_D to calculate the HMAC of the challenge $Hash(C_D, K_D)$. The debugger then forwards its computed HMAC to the module. The module compares the hashes and unlocks the JTAG instruction register in case of a match. In addition, the module forwards the tag ID of the authenticated debugger to the signal filters. If the HMAC comparison does not match, a LOCKED signal is returned to the debugger and the infrastructure remains locked.

In addition to providing the response of the authentication challenge, the HMAC mitigates potential man-in-the-middle attacks in case a network is used to interface the workstation and the SoC under debug. In such attacks, the adversary would leverage the network interface to either impersonate the debugger workstation or collect the communication between the workstation and the SoC and gain credentials of the debugger. By hashing the response using the private key of the debugger, the adversary cannot obtain the plain messages between the parties.

4.3. Asset Filtering

If a signal filter can trace assets, it is enhanced with security features to check when an asset is being traced, compare the tag of that asset to that of the authenticated debugger, and either expose or obfuscate the value of the trace. The key issue is to track where in SoC memory the assets are located at runtime, as they can be traced from different locations and by different filters. One way to solve this issue is to use dynamic information flow tracking, such as the one in Porquet and Sethumadhavan [2013], to propagate the tags. However, such a method requires changes to IP cores and cannot propagate tags of firmware assets. Instead of propagating the tags, we use a centralized method to monitor when assets are accessed at runtime, dynamically update information about their locations, and make this information available to all signal filters. Next we describe our method for data and firmware assets.

4.3.1. Monitoring Data Assets. We consider a cryptographic key K_{ex} to show how our method monitors data assets. Figure 6 provides one illustration of how K_{ex} could be accessed at runtime. On system startup, K_{ex} is stored in nonvolatile memory at address 0xF00. At runtime, the software makes a read memory request to address 0xF00 via the system fabric ①; the latter sends the address to the bus slave ②, and K_{ex} is returned to the fabric ③. K_{ex} is forwarded to the CPU, and the request is serviced ④. Later during execution, the software writes K_{ex} to the key register of the AES core at address

Fig. 6. Illustration of accessing a data asset at runtime.

Fig. 7. Monitoring data read requests.

0x800 (5), and the fabric writes K_{ex} at the given address (6). The AES core computes an encryption using the key, and the result is read out by the CPU (not shown).

An ideal method is to search the value of each data request in the tag LUT to see if it is an asset and to forward its tag to the signal filter of the relevant CUD. However, the tag LUT does not store the values of data assets by default. This is because these values may need to be provisioned after manufacturing (i.e., during OEM verification) or when the SoC is in the field due to compromised (i.e., leaked) assets [Khosravi et al. 2014]. Instead, our method obtains the value of an asset the first time it is requested at runtime (from its nonvolatile memory), monitors subsequent requests related to that asset, and updates information about its location. For each instance (i.e., copy) of an asset that is in use, our method stores an $\langle \text{address, value, tag} \rangle$ tuple in a volatile LUT that is accessible to the signal filters. This LUT is referred to as the *runtime LUT*.

Monitoring data read memory requests. For each read request, the method searches the tag and runtime LUTs for the requested address. The searches can find the following:

- (1) *No match in either LUT*: The address does not store an asset and is dismissed.
- (2) *Match only in the tag LUT*: The request is to read a data asset for the first time. This is because if the asset was already read, its address would match in the runtime LUT. For this match, the method allocates a new entry in the runtime LUT to hold the address and data of the request and the tag of the asset.
- (3) *Match in runtime LUT only*: The request is to read an asset that is already in use. The method dismisses the request because it does not copy or modify the asset.
- (4) *Match in both LUTs*: The method dismisses the request because it is a read for an asset that is already in use.

Figure 7 illustrates how the method works for reading our key K_{ex} example. When the CPU issues the read request for address 0xF00 at (1), the method searches the

Fig. 8. Monitoring data write requests.

tag and runtime LUTs for the address 0xF00. A match is found in the tag LUT only, illustrating case (2) of the four preceding outcomes. The method allocates a new entry in the runtime LUT (entry 0) and adds the address 0xF00 of the request and the tag 0100 of the asset in that entry. When the read request is serviced at ④, the method adds the data of the asset K_{ex} in the LUT entry.

Monitoring memory data write requests. For each write request, the method searches the runtime LUT for the address and data of the request. There is no need to search the tag LUT, as the SoC security architecture would prohibit illegal modifications of assets to nonvolatile memory. The runtime LUT search can find the following:

- (1) *No match for either search category:* The request is dismissed.
- (2) *Match only for the data search:* The request is to write a data asset to memory (e.g., a memory-mapped register). The method allocates a new entry in the runtime LUT for the address, data, and tag of the asset. The first two fields are obtained from the request itself, whereas the tag is from the matching entry in the LUT.
- (3) *Match only for address search:* The request is to write to a memory address that currently holds an asset. However, the request is writing data that has no confidentiality requirement, as that data does not match in the LUT. The method thus invalidates the matching entry in the runtime LUT because it will no longer contain an asset after the write.
- (4) *Matches for both address and data searches:* The request is to write to a memory address that currently holds an asset. The match in the runtime LUT indicates that the data of the request is also an asset. This can be the case that the same asset is being copied onto itself, or the more likely case that an asset is being overwritten by a new asset (i.e., in a cryptographic key register). In either case, the method obtains the asset tag from the LUT entry that matches the data and invalidates that entry. The method then updates the LUT entry that matches the address with the data of the write request and the obtained tag.

Figure 8 illustrates how the method handles the write request for the K_{ex} asset. At ⑤, a write request is issued and the method searches the runtime LUT for the address 0x800 and the data K_{ex} of the request. A match is found for the data search, and the method allocates a new runtime LUT entry (entry 1 in Figure 8) for the asset. At this point, the runtime LUT has information about each instance of the asset K_{ex} .

Leveraging runtime LUT for data tracing. We now explain how a signal filter uses the runtime LUT to check if the data being traced is an asset. For each data to trace, the filter searches the runtime LUT for the address of that data. If a match is found, the associated tag is returned. The filter then compares the tag to that of the current debugger to determine if the trace value should be obfuscated or exposed.

Fig. 9. Illustration of a secure signal filter using the runtime LUT.

- (1) *CPU or system fabric as CUD.* Figure 9 illustrates the interaction between the signal filter of the CPU and the runtime LUT during debug. This illustration is the same if the system fabric is the CUD. At ④ and ⑤, the signal filter traces the read and write requests, respectively, for K_{ex}. At ④, the filter searches the runtime LUT for a match of the address 0xF00. The LUT returns the associated tag 0100. The filter can then compare this tag to the tag ID of the authenticated debugger before tracing K_{ex}. At ⑤, the filter searches the runtime LUT for the address 0x800. The LUT returns the tag, and the filter can determine if K_{ex} should be exposed as a trace.
- (2) *AES as CUD.* Given that the AES key register is memory mapped, the filter can use the static address of the register (0x800 in our case) to access the runtime LUT and retrieve the tag of the asset. The filter then determines if the asset should be exposed or obfuscated based on the tag ID of the authenticated debugger.

4.3.2. Monitoring Firmware Assets. We use the same concept of the runtime LUT to build a method for firmware assets. A proprietary firmware is executed either in its base nonvolatile memory (e.g., bootROM) or is copied via direct memory access (DMA) to a memory segment where it is then executed. Our method considers both cases.

Monitoring instruction fetches. For each instruction fetch from the system fabric, the method checks if the address is in the tag and runtime LUTs. If a match is found in the tag LUT only, then a proprietary firmware is being executed from its nonvolatile memory. The method allocates a new entry in the runtime LUT to hold the base address, maximum address, and tag of the asset. The maximum address is obtained by adding the base address to the size mask from the tag LUT. Any other result of the LUT searches is dismissed.

Monitoring DMA transfers. The method monitors the SoC direct memory access controller (DMAC) for DMA transfers. The method searches the tag LUT using the source address of a DMA transfer and searches the runtime LUT using the destination address. The searches can find the following:

- (1) *No match:* The DMA transfer is not related to an asset and is dismissed.
- (2) *Match only in the tag LUT:* The DMA transfer copies a firmware asset from its nonvolatile memory to another memory segment. The method allocates a new entry in the runtime LUT to hold the base address, maximum address, and tag of the asset.
- (3) *Match found only in the runtime LUT:* The DMA transfer overwrites a firmware asset with a firmware (or data) that has no confidentiality requirement; the method invalidates the matching entry in the LUT.
- (4) *Matches found in both LUTs:* The DMA transfer overwrites a firmware asset with another firmware asset. The method thus overwrites the matching entry in the

Fig. 10. Secure debug controller.

runtime LUT with the base address, maximum address, and tag of the newly copied firmware asset.

Leveraging Runtime LUT for firmware tracing. We now explain how a signal filter leverages the runtime LUT to determine firmware assets.

The system fabric can trace instruction disassembly of a proprietary firmware via memory transfer tracing, and the CPU can trace the branch target addresses of a proprietary firmware via instruction tracing. For each trace of a memory transfer or a branch address, the signal filter of the CUD searches the runtime LUT for the address of the memory transfer or the branch target address. The runtime LUT returns a match if the address is in the range of the base and maximum addresses of an entry. The filter gets the tag of the entry and compares it to that of the debugger to expose or obfuscate the trace.

4.3.3. Putting It All Together. A secure debug controller (SDC), shown in Figure 10, encapsulates the data and firmware monitoring methods. The SDC receives relevant control signals for data requests and instruction fetches via the system fabric, and for DMA bulk transfers via the DMAC. The SDC also receives the addresses of data and instructions being traced from the secure signal filters and outputs the tags of assets. Figure 10(b) details the inner workings of the SDC. The data monitor uses control signals from the system fabric to detect memory requests relevant to data assets and update the runtime LUT accordingly. The firmware monitor uses the signals from the fabric and from the DMAC to update the runtime LUT when fetches or DMA transfers are relevant to firmware assets. The SDC uses an interface (SDC interface) to allow the signal filters to search the runtime LUT using the addresses of data and firmware being traced. The interface returns the tags associated with these addresses if they are data and firmware assets.

5. IMPLEMENTATION

There are several challenges to the implementation of our proposed security features, namely: (1) the SDC data and firmware monitors must check all data memory requests, instruction fetches, and DMA transfers; (2) the runtime LUT must have enough entries to avoid false positives; and (3) the runtime LUT must not become a delay bottleneck when accessed by multiple filters simultaneously.

Fig. 12. Implementation of the SDC data monitor.

5.3. Secure Debug Controller

5.3.1. Data Monitor. The data monitor is based on the AMBA AHB 2 protocol. The protocol defines a two-stage pipelined system fabric for memory requests. A request can have one (nonsequential) or multiple (burst/sequential) transfers. Each transfer has two stages: a single-cycle address stage when control signals (address, read/write, size, protection, etc.) are set and a data stage when the transfer is serviced. For transfers of a burst request, the values of most control signals remain the same, except for signals of the address, data value, and status of the request. For the purpose of implementation, we do not consider cases of failed transfers. Figure 12(a) gives a high-level view of the data monitor, which consists of three components: an address monitor to detect when a new request is issued, a data collector to store the data of the request, and a data checker to verify if the request is relevant to a data asset and to update the runtime LUT. The data collector stores information (type, address, data) about each request in a FIFO queue, and the checker reads from the queue.

Figure 12(b) details the state diagram of the address monitor. The monitor uses control signals of the AMBA AHB 2 fabric to determine if the current clock cycle is the address stage of a nonsequential transfer, indicating a new request. The monitor uses two equations: $NewReq$ and $Burst$. $NewReq$ indicates that a new request is issued and the fabric is ready to service it in the next cycle (i.e., data stage). This equation is true when the signal $HTRANS = b'10$ to indicate a nonsequential transfer, when $HREADY = b'1$ to acknowledge that the fabric is idle, when $HPROT[0] = b'1$ to indicate a data access, when $HMASTER \neq b'1111$, where $b'1111$ is the master ID of the DMAC for our implementation, and when $HSIZE[2:1] \neq b'11$ to ensure that the request is not for data greater than 256 bits. $Burst$ indicates a sequential transfer and is true when $HTRANS = b'11$. Given the pipelined nature of the AMBA AHB 2 fabric, the monitor checks $NewReq$ every cycle. In case of a new request, the monitor notifies the data collector, sends the address and type (read or write) of the request (obtained from $HADDR$ and $HWRITE$ signals), and goes

Fig. 13. Implementation of an SDC firmware monitor.

to a new request state. In case of a sequential transfer, the monitor goes from the new request state to a busy state.

Figure 12(c) illustrates the state diagram of the data collector. When notified of a new request, the collector goes to a new request state and obtains the data value of the request from the HHRDATA signal in case of a read or from HWDATA in case of a write. The data collector then checks if the Burst equation is true and transitions to a burst state to continue collecting the data of the request. When the request is complete, the data collector adds the address, type, and data of the request to the FIFO queue.

Figure 12(d) details the work of the data checker. The checker reads the oldest entry in the FIFO and checks if it is valid; this is indicated by a bit that is raised by the data collector when adding a new request. If the entry is valid, the checker reads the information for the request, searches the tag and runtime LUTs for a match of the request address, and searches the runtime LUT for a match of the request data. All LUT searches are performed simultaneously. Based on the request type (read or write), the data checker can either allocate, overwrite, or invalidate a runtime LUT entry as detailed in Section 4.3.1. Once the checker updates the runtime LUT for a request, it invalidates the FIFO entry for that request and goes to the next index of the FIFO.

5.3.2. Firmware Monitor. Figure 13(a) shows a high-level view of the firmware monitor, which consists of three components: an instruction fetch (IF) monitor to store information about instruction fetches, a DMAC monitor to store information about bulk transfers via the SoC DMA, and a firmware checker to update the runtime LUT when instruction fetches and DMA transfers are related to firmware assets. The IF monitor is similar to the address monitor in Figure 12(b). The only difference is the equation for NewReq, where $HPR0T[0] = 0$ for instruction fetches, and HSIZE is not considered because the processor may fetch instructions at the cache block granularity or 32 bits at a time. The IF monitor adds the address of each instruction fetch to its FIFO queue.

We use the DMAC from Gaisler et al. [2015b] for reference. A DMA transfer has two stages: a memory-to-buffer (M2B) stage, where the DMAC copies content from memory to its internal buffer, and a buffer-to-memory (B2M) stage, where the DMAC writes its buffer content to memory. For simplicity, we assume that an M2B stage is always followed by a B2M stage. The DMAC monitor uses the control signals of the DMAC to detect the two stages. Figure 13(b) shows the state diagram for the DMAC monitor, which uses two equations: M2B and B2M. M2B is true when the DMAC control signal Grant = 1 to indicate that the DMAC can initiate transfer, when Burst = 1 to indicate that the DMA transfer is for a burst request, and write = 0 to indicate that data is copied from memory to the DMAC buffer. The B2M equation is similar to M2B, except write = 1, which indicates that the data is copied from the DMAC buffer to SoC memory. When in the M2B state, the DMAC monitor records the address of the content

loaded from memory. When in the B2M state, the monitor records the address being written to and writes the M2B and B2M addresses in its FIFO.

The firmware checker uses round-robin priority to read the FIFO entries of the IF and DMAC monitors. If the FIFO with highest priority is empty, the checker goes to the other one. When verifying an instruction fetch, the checker simultaneously searches the tag and runtime LUTs for a match of the address of the fetch. When verifying a DMA transfer, the checker simultaneously searches the tag LUT for the B2M address and the runtime LUT for the M2B address. The checker updates the runtime LUT as detailed in Section 4.3.2. The checker then invalidates the FIFO entry for the verified instruction fetch or DMA transfer and goes to the next index of the FIFO.

5.3.3. LUTs. The tag LUT is implemented as a read-only nonvolatile content addressable memory (CAM). The LUT has two read ports (one for each monitor). Each entry is 69 bits, where the least significant 4 bits indicate the tag of the asset, the next bit indicates if the asset is a data (0) or a firmware (1), the next 32 bits are for the mask size of the asset, and the last 32 bits are for its base address. To put less pressure on the SDC during reads and writes, the runtime LUT is implemented as two components: one for data and the other for firmware assets. Each component is as a fully associative cache. For the data component, each entry has a 32-bit address field, a 256-bit data field, and a 4-bit tag field. This component of the runtime LUT has two read ports, to allow simultaneous searches by address and data, and one write port. For the firmware component, each entry has a 32-bit base address field, a 32-bit maximum address field, and a 4-bit tag field.

One major implementation issue is the size of each runtime LUT component. It must be large enough to store information about all assets accessed at runtime to avoid false positives/negatives. We configure each component with twice as many entries as the tag LUT. This is based on the reasoning that an asset is unlikely to be in more than two locations at runtime: its nonvolatile base memory segment and another volatile memory segment in the SoC when in use. In addition, the runtime LUT is optimized to invalidate entries when their associated assets are no longer in use.

5.4. FPGA Prototype

We implement a prototype of the baseline and security-enhanced debug infrastructure on a Xilinx Spartan-3 FPGA. The baseline prototype is similar to the SoC in Figure 1(b), and the security-enhanced one incorporates the SDC and its connections to the DMAC and the system fabric, as shown in Figure 14. The CPU is the SPARC LEON3 processor obtained from Gaisler et al. [2015b]. The DMAC and the AMBA AHB 2 system fabric are also obtained from Gaisler et al. [2015b]. We augment a standard AES IP core with an AMBA AHB slave interface. For prototyping, we set the number of assets to four. The signal filters are implemented as buffers that store relevant trace signals and overflow when full. Timing reports from synthesis of the prototypes show that they run at a maximum frequency of 50MHz.

The correctness of our approach depends on the ability of the SDC to verify all memory requests, instruction fetches, and DMA transfers. This way, the runtime LUT will be updated correctly, and the signal filters can trivially obtain the tags of assets being traced. As Figures 12(a) and 13(a) show, both data and firmware monitors are pipelined. Using Figure 12(a) as an example, the pipeline has three stages for the address monitor, data collector, and data checker. The bottleneck of the pipeline is the number of cycles to access the LUTs. For our prototype, we assume that each LUT access takes one clock cycle. We thus set the FIFO of the SDC data monitor with four entries. The same size is set for the SDC firmware monitor.

Fig. 14. Secure trace-based debug infrastructure.

We validate the correctness using MiBench benchmarks [Guthaus et al. 2001]. We modify the benchmarks to include their reference inputs in the source code and cross compile them for SPARC. We separate the SoC ROM into code and data segments, where the code is considered proprietary firmware and the data segment stores three cryptographic keys for different asset owners. We modify the benchmarks to read the keys at different stages of execution. Using the synthesized model of the security-enhanced prototype, we run each benchmark and track the information (data, address, type) of each memory data request, as well as the address of each instruction fetch going through the system fabric. We also track the data memory requests and instruction fetches verified by the data and IF monitors of the SDC. Our comparisons confirm that for our evaluated benchmarks, the SDC successfully monitors all data memory requests and instruction fetches, and it updates the runtime LUT accordingly. For the DMA monitor, we use a test bench that issues a 1KB DMA transfer every 10 clock cycles after completion of the previous one. Our results confirm that the firmware checker correctly verifies all DMA transfers and updates the runtime LUT.

5.5. ASIC Evaluation

We evaluate the area and power overheads of our security features on an ASIC. We assume that the SoC has 32 assets. Each component in the runtime LUT thus has 64 entries. We assume a 1GHz SoC. We augment NVSim [Dong et al. 2012] to estimate the tag LUT as phase-change memory (PCM) CAM. We use Cacti [Muralimanohar et al. 2008] to estimate the components of the runtime LUT as fully associative caches. For the data component of the runtime LUT, we consider two caches: one with 32-bit cache tags for addresses of assets and the other with 256-bit cache tags for the values of assets. The runtime LUT component for firmware assets is also evaluated as two associative caches, each with 32-bit cache tags and 8-bit cache lines. One cache stores the base addresses of proprietary firmware, and the other holds the maximum addresses. Table II shows the configurations of the tag and runtime LUTs.

We consider the OMAP 4430 as a baseline reference SoC. The SoC has a dual-core ARM Cortex-A9 processor, a 3D graphics accelerator subsystem, and several other cores [Witt 2009]. Using McPAT [Li et al. 2009], we estimate the area and power costs of the processor subsystem to be 11.3mm² and 5.74W, respectively. We synthesize the authentication module, secure signal filter, and data and firmware monitors of the SDC using 45nm cell technology from FreePDK [Stine et al. 2007]. All components

Table II. Characteristics of Tag and Runtime LUTs for ASIC Implementations

	Size	Cache tag	Access Delay (ns)	Area (mm ²)	Power (mW)
Tag LUT	32×64	32	0.98	0.0179	12.43
Runtime LUT					
Data 1	64×8	32	0.39	0.033	37.8
Data 2	64×8	256	1.347	0.346	97.12
Firmware 1/2	64×8	32	0.39	0.033	37.8

Table III. Area and Power Costs of Secure Debug Components in an ASIC

	Area (mm ²)	Power (mW)
Authentication Module	0.037986	11.3508
Secure Debug Controller	0.4979	296.22
Secure Signal Filter	3.56×10^{-4}	0.4276

are configured to run at the 1GHz system clock, except for the authentication module, which runs at 100MHz. This is because the module is part of the JTAG port that traditionally runs at lower frequency than the system clock. Table III shows the area and power costs of the added components.

5.5.1. Area Overhead. We emphasize that only one authentication module and one SDC are needed for the SoC. The enhancements require 0.55mm², incurring 4.8% additional area when compared to the processor subsystem. When considering the 70mm² of the complete SoC [Blem et al. 2013], the overhead is less than 1%. The low cost of the enhancements allows the SoC integrator to consider them during architectural exploration of the design.

5.5.2. Power Overhead. Our enhancements increase the power consumption during debugger authentication, which only occurs once, and during runtime debugging due to the work of the SDC and the secure signal filters. During authentication, the HMAC consumes the most power—10.65mW. Since this authentication occurs during system boot-up, the rest of the SoC is mostly idle and this overhead is within the SoC power constraints. At runtime debug, the enhancements require 296.2mW, incurring 5.15%. We note this is a pessimistic overhead because it only considers the processor and does not include the graphics accelerator and other components. If we consider a conservative estimation of the full OMAP 4430 SoC, where the graphics accelerator has the same peak power as the processor and all other subsystems have a combined 1W peak power, the overhead would be 2%. Note that this overhead only occurs during debugging. During functional mode, the SDC and other added components are power gated.

5.5.3. Performance Overhead. Our enhancements do not delay SoC execution. The debugger authentication occurs during boot and has no impact on the software. During runtime debug, the SDC is able to monitor all transfers along the system fabric and DMAC, as long as its work is pipelined as described in Section 5.4 and buffers in the SDC are sufficient to mask the access delays in the LUTs. As Table II indicates, most components of the LUTs require one cycle per access. The only exception is the data 2 component of the runtime LUT, which requires two cycles. Given that this component needs to be accessed by the data checker and collector stages of the data monitor, a second read port can be added to avoid any structural hazards between the two stages.

Fig. 15. Broadcasting method to synchronize local runtime LUTs of signal filters.

6. DISCUSSIONS AND LIMITATIONS

6.1. Potential Attacks on Secure Debug Infrastructure

We discuss several approaches that an adversary may use to bypass our secure debug infrastructure and mitigation mechanisms.

Bypassing secure asset tagging. A malicious asset owner may falsify the address and size mask of one of its assets to conflict with that of another owner. However, the integrator receives the address and size of each asset and can detect such conflicts.

Bypassing debugger authentication. Replay attacks that monitor each authentication challenge C_D will not succeed because a challenge is chosen as a 128-bit pseudorandom number. The integrator can use a low-cost true random number generator [Srinivasan et al. 2010] to further mitigate such attacks. Although the adversary can pass the tag ID of an authorized asset owner, the authentication will not be successful without the key K_D of that owner. If data remanence attacks in the ROM are a concern, the integrator can use a PUF-based authentication such as that in the work of Das et al. [2012].

Leaking assets via LUTs. The runtime LUT is not exposed to the debugger via an external port (e.g., JTAG) or through software. Moreover, this LUT is volatile and does not retain the data assets on power off. The adversary thus cannot read assets through that LUT. The tag LUT does not store assets and thus cannot be used to leak them.

Modifying configuration of secure signal filters. An adversary may try to modify the tag ID register of a signal filter via the JTAG port. This can be thwarted by setting this register as a write-once register that is set by the authentication module.

6.2. Scalability of the SDC Interface

If several secure signal filters are added to the debug infrastructure, each needs to be independently connected to the SDC. This requires multiple read ports for the SDC, delaying the access time to the LUTs. For example, with 16 secure signal filters (e.g., an SoC with 16 debuggable IP cores), the runtime LUT takes more than five cycles per read access.

To overcome this delay, each secure debug filter can be enhanced with a copy of the runtime LUT. This way, the filter can access the LUT locally during tracing, incurring a one-cycle delay. This requires the SDC to synchronize the (remote) runtime LUTs of the signal filters when it has updated its own runtime LUT. Figure 15 shows one design for such synchronization. A lightweight, broadcasting single-master, multislave fabric, such as the AMBA AHB 3-Lite [ARM 2006], connects the SDC to the secure signal filters. Each remote runtime LUT consists of the data 1, firmware 1, and

firmware 2 components detailed in Table II. In addition, each secure signal filter has a memory-mapped synchronization register to receive updates from the SDC. When the SDC updates an entry in its runtime LUT, its interface issues a write request to the synchronization register of each active signal filter using the ADDR and DATA signals of the fabric. The broadcast decoder then uses the HSEL to select the correct filters. Note that the SDC may need an encoding mechanism to indicate if the synchronization is for invalidation, allocation, or update of an LUT entry. This can be done by sending two transfers within the write request, where the first transfer encodes the purpose of the synchronization and the second one contains the new information (e.g., address of the newly accessed asset). We also note that although this approach increases the area of a filter by almost $300\times$ (to 0.09mm^2), it is a scalable solution because each filter is associated with an IP core and incurs negligible area cost on that IP.

6.3. Adherence to Secure Debug Requirements

Our infrastructure meets the secure debug requirements detailed in Ray et al. [2015]:

- (1) *HVM*: Our approach does not change the execution flow of firmware or modify data. Assets are not obfuscated until they are being traced. This way, the functional execution during debug is consistent regardless of the debugger.
- (2) *Reusability*: By restricting access at the asset granularity and automating asset filtering at runtime, our approach can consider all use cases of an asset based on its access policy. Our approach is adaptable for different SoC products, as the integrator simply needs to update the tag LUT to reflect new assets or new access policies.
- (3) *Late variability*: Our approach is not limited by new signals added for debug. If a new asset-relevant signal is added, the integrator simply enhances its signal filter with the security features in Figure 11(b). These enhancements require less than $10\times 10^{-3}\text{mm}^2$ and do not impact the debug infrastructure.
- (4) *Self-securability*: Our approach does not present new backdoors (see Section 6.1).
- (5) *Architecture*: The SDC is a centralized IP for security and can be efficiently validated. In addition, it incurs negligible area and power overheads, which allows integrators to consider its practicality early in the SoC design phase.

6.4. Compatibility of Approach with Other Debug Infrastructures

Although we use the ARM CoreSight debug infrastructure for illustration, our proposed security features can be adapted for other infrastructures. This is because our features do not rely on inherent characteristics of the ARM CoreSight or ARM architecture (e.g., AMBA fabric) to provide security. More specifically, the debugger authentication module can be adapted to any JTAG or debug access port; based on the generic description in Section 4.3, the SDC can be implemented for different system fabric protocols to allow dynamic monitoring of assets; and the modifications to the signal filters for asset obfuscation are negligible (Muxes and tag ID register) and can be adapted to any hardware components that perform such trace filtering.

6.5. Unmonitored Assets

Our tagging approach relies on the addresses of assets in memory. As a result, our enhanced debug infrastructure can only secure memory-mapped assets such as data and firmware stored in nonvolatile memory and memory-mapped registers. Assets that are not memory mapped cannot be protected by our approach.

7. CONCLUDING REMARKS

The backdoor exposed by trace-based debugging is a critical security vulnerability because (1) untrusted debuggers can access the trace-based debug infrastructure, (2) attacks that leverage debug traces do not require side-channel analysis or in-depth knowledge of the SoC, and (3) adversaries can launch such attacks at different stages of the SoC life cycle. We present a secure trace-based debugging infrastructure to shut this backdoor. We add practical security features to enforce the confidentiality of assets in real time and without impacting the debugging process. Our FPGA prototype confirms the functional correctness of our design and implementation, and our ASIC evaluations show that the security enhancements incur 5% area and power overheads.

REFERENCES

- ARM. 1999. AMBA Specifications (Rev 2.0). ARM Limited.
- ARM. 2006. AMBA 3 AHB-Lite Protocol (v1.0). ARM Limited.
- ARM. 2008. AMBA AHB Trace Macrocell Technical Reference Manual. ARM Limited.
- ARM. 2009. Security Technology Building a Secure System Using TrustZone Technology. ARM Technical White Paper.
- ARM. 2011. CoreSight Program Flow Trace PFTv1.0 and PFTv1.1: Architecture Specification. ARM Limited.
- ARM. 2013. ARM CoreSight Architecture Specification v2.0. ARM Limited.
- E. Ayday and S. Rajagopal. 2011. Secure, intuitive and low-cost device authentication for smart grid networks. In *Proceedings of the IEEE Consumer Communications and Networking Conference*. 1161–1165.
- E. Blem, J. Menon, and K. Sankaralingam. 2013. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 1–12.
- G. Chiu and J. Li. 2012. A secure test wrapper design against internal and boundary scan attacks for embedded cores. *IEEE Transactions Very Large Scale Integrated Systems* 20, 1, 126–134.
- J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. 2005. SECA: Security-enhanced communication architecture. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. 78–89.
- A. Das, U. Kocabas, A. Sadeghi, and I. Verbauwhede. 2012. PUF-based secure test wrapper design for cryptographic SoC testing. In *Proceedings of the Design, Automation, and Test in Europe Conference Exhibition*. 866–869.
- X. Dong, C. Xu, Y. Xie, and N. Jouppi. 2012. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7, 994–1007.
- J. Dworak, A. Crouch, J. Potter, A. Zygmuntowicz, and M. Thornton. 2013. Don't forget to lock your SIB: Hiding instruments using P1687. In *Proceedings of the IEEE International Test Conference*. 1–10.
- J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc. 2015b. *GRLIB IP Core Users Manual*. Gaisler Research, Goteborg, Sweden.
- J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc. 2015a. *GRLIB IP Library Users Manual*. Gaisler Research, Goteborg, Sweden.
- T. Goodspeed and A. Francillon. 2009. Half-blind attacks: Mask ROM bootloaders are dangerous. In *Proceedings of the USENIX Conference on Offensive Technologies*. 6.
- M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*. 3–14.
- D. A Hardy, F. Sydnor, and W. Oh. 2010. Method for managing operability of on-chip debug capability. United States Patent 7,849,315.
- Intel. 2015. *Intel Trace Hub-Developer's Manual*. Intel, Santa Clara, CA.
- R. Johnson, M. Rubnich, and A. DelaCruz. 2011. Implementing a key recovery attack on the high-bandwidth digital content protection protocol. In *Proceedings of the IEEE Consumer Communications and Networking Conference*. 313–317.
- H. Khosravi, P. Munguia, A. Pearson, S. Brown, and D. Schollmeyer. 2014. Secure remote credential provisioning. United States Patent 8,645,677.

- S. Lau and V. Varshney. 2014. Dynamic device identification for making a JTAG debug connection with a internet browser. United States Patent 8,671,319.
- J. Lee, M. Tehranipoor, and J. Plusquellic. 2006. A low-cost solution for protecting IPs against scan-based side-channel attacks. In *Proceedings of the IEEE VLSI Test Symposium*. 94–99.
- S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. 469–480.
- X. Liu and Q. Xu. 2009. Trace signal selection for visibility enhancement in post-silicon validation. In *Proceedings of the Design Automation, and Test in Europe Conference Exhibition*. 1338–1343.
- N. Muralimanohar, R. Balasubramanian, and N. Jouppi. 2008. Architecting efficient interconnects for large caches with Cacti 6.0. *IEEE Micro* 11, 69–79.
- ODA. 2016. Online Disassembler. Retrieved November 6, 2016, from <https://www.onlinedisassembler.com/static/home/>
- W. Orme. 2008. Debug and Trace for Multicore SoCs. ARM White Paper.
- L. Pierce and S. Tragoudas. 2013. Enhanced secure architecture for joint action test group systems. *IEEE Transactions on Very Large Scale Integration Systems* 21, 7, 1342–1345.
- J. Porquet and S. Sethumadhavan. 2013. WHISK: An uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 4:1–4:9.
- S. Ray, Jin Yang, A. Basak, and S. Bhunia. 2015. Correctness and security at odds: Post-silicon validation of modern SoC designs. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1–6.
- K. Rosenfeld and R. Karri. 2010. Attacks and defenses for JTAG. *IEEE Design and Test of Computers* 27, 1, 36–47.
- S. Srinivasan, S. Mathew, R. Ramanarayanan, F. Sheikh, M. Anders, H. Kaul, V. Erraguntla, R. Krishnamurthy, and G. Taylor. 2010. 2.4GHz 7mW all-digital PVT-variation tolerant true random number generator in 45nm CMOS. In *Proceedings of the IEEE Symposium on VLSI Circuits*. 203–204.
- J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavara-jaiiah, J. Oh, and R. Jenkal. 2007. FreePDK: An open-source variation-aware design kit. In *Proceedings of the IEEE International Conference on Microelectronic Systems Education*. 173–174.
- L. Uhsadel, A. Georges, and I. Verbauwhede. 2008. Exploiting hardware performance counters. In *Proceedings of the IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography*. 59–67.
- B. Vermeulen and K. Goossens. 2014. *Debugging Systems-on-Chip: Communication-Centric and Abstraction-Based Techniques*. Springer.
- D. Witt. 2009. OMAP4430 architecture and development. In *Proceedings of the IEEE Hot Chips 21 Symposium*. 1–16.
- B. Yang, K. Wu, and R. Karri. 2004. Scan based side channel attack on dedicated hardware implementations of data encryption standard. In *Proceedings of the IEEE International Test Conference*. 339–344.
- X. Zhang, O. Aciicmez, and J. Seifert. 2007. A trusted mobile phone reference architecture via secure kernel. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*. 7–14.

Received March 2016; revised July 2016; accepted September 2016