Exploiting Design-for-Debug for Flexible SoC Security Architecture

Abhishek Basak
Dept. of EECS
Case Western Reserve Univ.
Cleveland, OH, USA
axb594@case.edu

Swarup Bhunia
Dept. of ECE
University of Florida
Gainesville, FL, USA
swarup@ece.ufl.edu

Sandip Ray
Strategic CAD Labs
Intel Corporation
Hillsboro, OR, USA
sandip.ray@intel.com

ABSTRACT

Systematic implementation of System-on-Chip (SoC) security policies typically involves smart wrappers extracting local security critical events of interest from Intellectual Property (IP) blocks, together with a control engine that communicates with the wrappers to analyze the events for policy adherence. However, developing customized wrappers at each IP for security requirements may incur significant overhead in area and hardware resources. In this paper, we address this problem by exploiting the extensive design-for-debug (DfD) instrumentation already available on-chip. In addition to reduction in the overall hardware overhead, the approach also adds flexibility to the security architecture itself, *e.g.*, permitting use of on-field DfD instrumentation, survivability and control hooks to patch security policy implementation in response to bugs and attacks found at post-silicon or changing security requirements on-field. We show how to design scalable interface between security and debug architectures that provides the benefits of flexibility to security policy implementation without interfering with existing debug and survivability use cases and at minimal additional cost in energy and design complexity.

1. INTRODUCTION

Today's embedded and mobile systems contain a significant amount of sensitive information and data (often collectively referred to as *assets*) that must be protected from unauthorized access. Such assets include private end-user information, cryptographic and DRM keys, firmware, debug and control modes, defeaturing bits, etc. Consequently, most system design specifications include a number of *security policies* [1, 2, 3] that define access constraints to these assets at different phases in the system execution. In current industrial System-on-Chip (SoC) design development, these policies are defined at different phases of design exploration, planning, and development by system architects as well as different IP and SoC integration teams, and often get refined or updated across the system development and validation phases. Unfortunately, the policies are highly subtle, involving multiple IPs and complex hardware, firmware and software interactions. Their implementations are intertwined with the implementation of the system functionality through a combination of RTL, firmware, or software components. Consequently, it is challenging to validate system adherence against a set of policies, or update the policies themselves post production, *e.g.*, in response to changing customer needs or newly identified attack scenarios.

Previous work attempted to address this problem by developing a flexible and generic security architecture for implementing SoC security policies [4, 5]. The architecture constitutes a centralized control engine programmed with the restrictions imposed by the various security policies. The system was shown to be flexible for implementing diverse security policies, including access control, time-of-check time-of-use (TOCTOU), and system boot policies. However, the architecture required each individual IP in the SoC to be augmented with a wrapper customized for security requirements; the wrappers identified the security-relevant events, communications, and data in the IP and coordinated with the centralized controller to ensure policy adherence. In a SoC design containing a large number of IPs, the cost incurred by such approach in terms of area overhead and design complexity may be prohibitive.

The key insight of this paper is that we can implement a security policy control system without incurring significant additional architecture and design overhead, by exploiting infrastructures already available on-chip. In particular, modern SoC designs contain a significant amount of Design-for-Debug (DfD) features to enable observability and control of the design execution during post-silicon debug and validation, and provide means to "patch" the design in response to errors or vulnerabilities found on-field. On the other hand, usage of this instrumentation post production, *i.e.*, for on-field debug and error mitigation, is sporadic and rare. Consequently, computing systems have a significant amount of mature hardware infrastructure for control and observability of internal events, that is available and unused during normal system usages.

Our main contribution is a flexible architecture that exploits on-chip DfD features for implementing SoC security policies. We show how to build efficient, low-overhead security wrappers by re-purposing the debug infrastructure, while being transparent to debug and validation usages. We illustrate some of the design trade-offs involved between complexity, transparency needs, and energy efficiency. Our experimental results on a SoC design with illustrative DfD

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898020

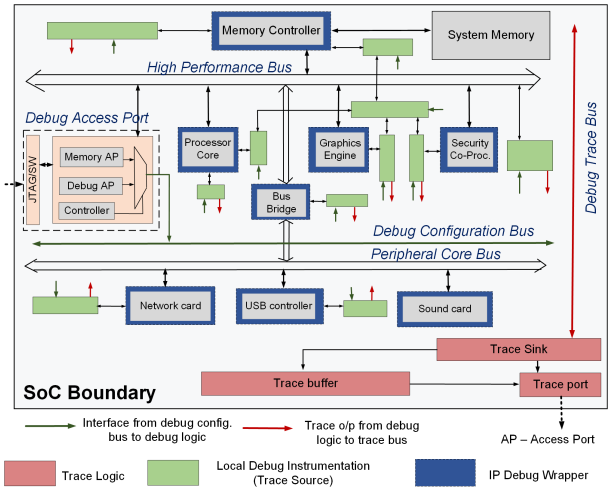

Figure 1: Simplified SoC DfD Architecture Based on Coresight™.

instrumentation shows that the approach could potentially provide significant savings in area and hardware overhead while permitting flexibility in on-field adaptation of security policies over a dedicated wrapper implementation, without significant increase in power/energy consumption.

The remainder of the paper is organized as follows. Section 2 provides the relevant background on DfD features, SoC security policies, and security architecture. Section 3 identifies the constraints involved in re-purposing DfD features. Section 4 describes some key components of our proposed security architecture. Section 5 defines some illustrative use cases. We provide experimental results on the overhead of the architecture in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. BACKGROUND

2.1 On-Chip Debug Infrastructure

Design-for-Debug (DfD) refers to on-chip hardware for facilitating post-silicon validation [6]. A key requirement for post-silicon validation is observability and controllability of internal signals during silicon execution. DfD in modern SoC designs include facilities to trace critical hardware signals, dump contents of registers and memory arrays, patch microcode and firmware, create user-defined triggers and interrupts, etc. Furthermore, DfD architecture is increasingly getting standardized to enable third-party EDA vendors to create software APIs for accessing and controlling the hardware instrumentation for system-level debug. As an example, ARM Coresight™ architecture [7] (Fig. 1) provides facilities for tracing, synchronization, and time-stamping hardware and software events, a trigger logic, and facilities for standardized DfD access and trace transport. The architecture is instantiated into *Macrocells* that can interact with IP functionality through standard interfaces. Debug interface is used not only for post-silicon but to enable workarounds for bugs and defects found on-field.

2.2 Security Policies and Security Architecture

Security policies govern access to sensitive assets in a SoC design. Following are two illustrative policies: [1, 4, 8]

1. During boot, keys transmitted by the crypto engine cannot be observed by any IP other than its intended target.

2. An on-chip fuse can be updated for silicon validation but not after production.

Previous work [4] defined a security architecture, called "E-IIPS" for disciplined, systematic implementation of such policies. The architecture includes a central security policy controller (SPC) that keeps track of the system security state and enforces the restrictions imposed by the policies in that state. SPC communicates with security wrappers for individual IP blocks that detect security critical events from IP operations.

3. DFD RE-PURPOSING CONSTRAINTS

DfD is a complex component of SoC design, that is developed to cater to the needs of validation, debug, and on-field workarounds. In this section we summarize some of the constraints and requirements that must be satisfied while re-purposing this architecture for security.

Transparency to Debug Use Cases. Post-silicon debug and validation are themselves critical activities performed under highly aggressive schedules. It is therefore critical that re-purposing the DfD does not interfere or "compete" with debug usages of the same hardware. For instance, if a trace or trigger module is necessary for a debug usage, then it cannot be simultaneously used for enforcing security constraints.

Maintaining Power-Performance Profile. On-chip instrumentation is optimized for energy and performance in usages related to debug. For example, since debug traffic is typically "bursty", it is possible to incur low penalty in power consumption even with a high-bandwidth fabric by power-gating the fabric components during normal execution; while re-purposing the same infrastructure, one must ensure that the power profile is not significantly disrupted by the new usages.

Acceptable Overhead for Interfacing Hardware. A key motivation for exploiting DfD for security architecture is reducing hardware overhead. However, this benefit would be obviously lost if significant hardware is necessary to interface with the DfD and configure it for security needs.

4. DFD-BASED SECURITY ARCHITECTURE

Our architecture is built on top of the E-IIPS framework [4]. We exploit DfD to implement smart security wrappers for IPs that communicate with the centralized policy controller (SPC), which implements security policies. Furthermore, SPC can program DfD to implement security controls using logic available for on-field upgrades.

4.1 Detection of Security-Critical Events

To avoid potential routing bottlenecks due to high bandwidth requirements, IP security wrappers must detect a set of security-critical events. The events necessary are based on the policy as well as the IP involved, *e.g.*, for a CPU we need to detect attempts of privilege escalation and monitor control flow of programs to detect probable presence of malware as well as prevent fine-grained timing based masquerading attacks. Developing IP security wrappers thus re-

Table 1: Security Critical Events detected by DfD Trace Cell in Processor Core

Trigger Event	*Ex. Security Context*
Prog. counter at specific address, page, address range	Prevent Malicious prog. trying to gain elevated privileges
System mode traps for specific interrupts, I/O, file handle,return from interrupt	Verify limited special register access by other IPs in kernel mode
High Branch Taken, Jump Instr. Frequency	Highly branched code often signs of malware
Invalid instruction opcode, Frequent div. by 0 exceptions	Un-trusted prog. source; Apply strict access cntrl.
Read/Write access to specific data memory page/s	Protect confidentiality, integrity of security asset
# of clock cycles between 2 events = threshold	Satisfy resource availability & avoid deadlock
More than one inter-communicating threads	Verify TOCTOU policy in firmware load in μC

quires custom logic for identification of this event set. However, the DfD modules already detect the information necessary for many of these events. Table 1 illustrates a few representative security-critical events that can be detected through Coresight™ macrocell for a typical processor core. Here, the macrocell is assumed to implement standard instruction and data value and address comparators, condition code/status flags match check, performance counters, event sequencers, etc. Correspondingly, macrocells for NoC fabric routers detect bulk of the critical events required for addressing threats such as malicious packet redirection, IP masquerade, etc.

4.2 Debug-Aware IP Security Wrapper

We architect smart security wrappers for each IP by exploiting DfD to identify relevant security-critical events; on detection of such an event, DfD communicates the information to the IP security wrapper that communicates it to the centralized SPC. To enable this functionality without hampering debug usages for the DfD, we need local (IP-level) modification of DfD logic and appropriate adaptation of the security wrapper. Fig. 2 illustrates the additional hardware requirements. In particular, noninterference with debug usage especially system energy/power constraints, requires transmission of security data to SPC via a separate port (instead of re-purposing the debug trace port and bus), which requires an additional trigger logic. The events of interest for the IP are programmed by the SPC via the configuration register interface of the corresponding DfD module. Since DfD module can be configured to detect a number of security events (related or disparate) at runtime, SPC must correctly identify the corresponding event from the communication frame sent by the security wrapper. We standardize this interface across all local DfD/security-wrapper pairs, by tagging event information with the corresponding configuration register address. Note that this standardization comes at the cost of additional register overhead in IPs where one or only a few events are detected via debug logic. Trace packet generation controls can be disabled during SPC access (resp., security-debug interface disabled during system debug) to save leakage power when debug (resp., security)

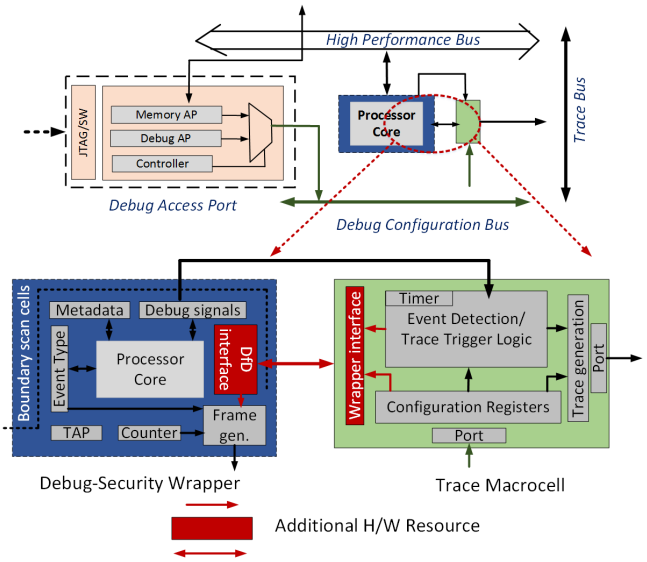

Figure 2: Additional hardware resources for interfacing local DfD with IP security wrapper.

architecture is not in use. Besides security critical event triggers and observability of associated information, the local DfD control hooks can also be re-purposed (by appropriate SPC configuration) to enforce security controls in the IP via the existing debug wrapper, during both design and on-field patch/upgrade phase.

4.3 SPC-Debug Infrastructure Interface

The Security Policy Controller (SPC) must be able to configure the individual local on-chip debug logic to detect relevant security-critical events and assert appropriate controls. Fig. 3 illustrates the communication between SPC and the debug interface. As used during system debug, the existing configuration bus (address and data) is used by SPC for trace cell programming. As there are usually enough configuration registers and associated logic in the local DfD components to monitor all possible security-critical events, the SPC can configure the trace cells with appropriate values at boot phase; therefore, the configuration fabric can be turned off during most times to save leakage power. In some rare scenarios, SPC cannot configure DfD for detection of all necessary events at boot; for these cases, SPC interfaces with the power management module to turn on the Debug Access Port (DAP) and configuration bus at runtime.

4.4 Design Methodology

A SoC design flow involves integration of a collection of (often pre-designed) IPs through a composition of NoC fabrics. An architectural modification involving communication among different IPs typically disrupts the SoC integration flow. We now outline the changes necessary to SoC integration to adapt to our proposed DfD-security architecture.

IP Designer: The IP provider needs to map security-critical events to the DfD instrumentation for the IP. The respective configuration register values are derived from the debug programming model. Finally, the security wrapper is augmented with custom logic for events not detected by DfD, and the standardized event frames for communicating

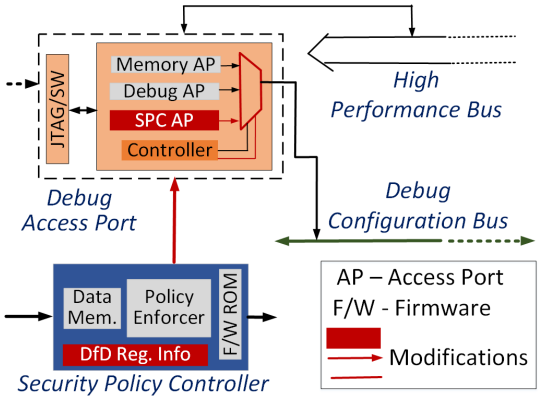

Figure 3: Interfacing SPC with on-chip debug.

with SPC are created, augmenting with DfD interface.

SoC Integrator: The SoC integrator augments the event detection logic of the local DfD instrumentation in IPs with triggers to the DfD/security-wrapper interface for transmission of event occurrence information, modifies debug access port incorporating SPC access, and adds necessary security and debug access control requirements to ensure debug transparency in the presence of security requirements. For the latter use case scenario *i.e.*, ensuring security during debug, where the DfD may not be re-purposed to detect all security critical events, the SoC integrator may proceed with more proactive, stricter security controls for SoC operations during debug. This includes incorporating coarse-grained policies not requiring all of the corresponding event information to transition to a new security state. All the necessary configuration register addresses/values are stored in the additional data memory/buffer in SPC to uniquely identify DfD detected security critical events. The SPC is also augmented with firmware instructions to configure these debug registers, mostly during boot.

5. USE CASES

5.1 An Illustrative Policy Implementation

To illustrate the use of our framework, we consider its use in implementing the following illustrative policy:

I/O Noninterference. When the CPU is executing in high-security mode, I/O devices cannot access protected data memory.

The policy, albeit hypothetical, is illustrative of representative security requirements involved in protecting assets from malicious I/O device drivers. Fig. 4 illustrates the flow of events involved in the implementation of this policy by SPC through DfD/security-wrappers. Here the DfD configuration is through a debug access port, CPU has an associated Embedded Trace Macrocell (ETM), and I/O device requests are assumed to be based on Direct Memory Access (DMA). Following are the key steps involved.

1. During boot, SPC configures the required DfD instruments. This includes "*program counter within secure code range*" and "*write access to protected data memory*" event triggers in the ETM.

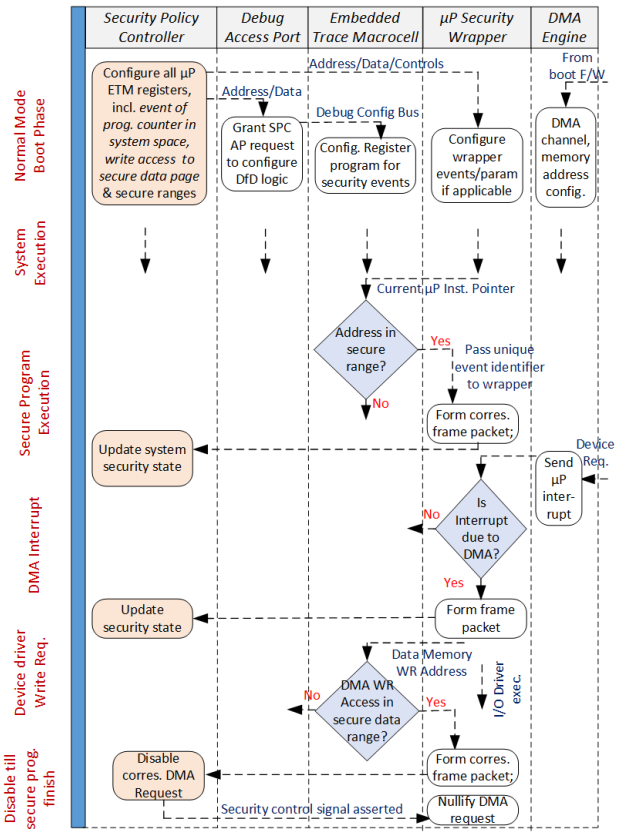

Figure 4: Use case scenario of security policy implementation exploiting the local DfD instrumentation.

2. Along with DfD, the SPC configures the IP security wrappers for the subset of events to detect, frame protocol to follow, scan chain access etc. DMA engines are also configured by boot firmware on device-channel and channel to memory mapping.

3. When a secure program is loaded, the ETM detects the event, and triggers the security wrapper which communicates with SPC. The SPC updates the security state.

4. A DMA interrupt is detected by the corresponding security wrapper and transmitted to SPC. Any write request from the high-privilege driver to the protected memory is detected by ETM and transmitted to SPC via wrapper. The SPC identifies policy violation from the current security state and enforces controls.

5.2 On-Field Policy Implementation/Patch

Given changing security requirements (*e.g.*, adoption of system in different market segments) or to address bugs or attacks detected on-field, new policies may need to be implemented or existing ones need to be upgraded or patched. These may require new events to be detected (outside what had been considered in design phase), extraction of more event information, and control the IP functionality in response to a policy. The interface of the security architecture with DfD allows the possibility of on-field system reconfiguration and upgrade, — something virtually impossible to perform with security wrappers customized for specific

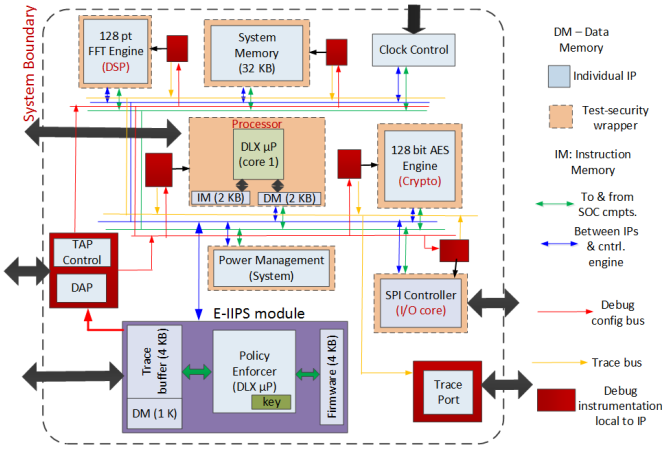

Figure 5: Block diagram schematic of SoC model with on-chip debug infrastructure.

security policies. Achieving this requires selection of local DfD at different IPs to identify if the relevant events can be detected; if so, the corresponding register address and value are added to SPC memory to be sent through DAP at boot/execution. With a standardized DfD interface with the IP security and debug wrappers, the corresponding events can be uniquely detected and control signals asserted in the IP if applicable.

6. EXPERIMENTAL RESULTS

Due to lack of standard open-source models of studying SoC architecture, we have been developing our own SoC design model. Although simpler than an industry design, our model is substantial and can be used for implementing realistic security requirements. Fig. 5 shows the current version of our model. It includes a DLX microprocessor core (DLX) with code memory, a 128b FFT engine (FFT), a 128b AES crypto core (AES), and a SPI controller. The IPs are augmented with security wrappers according to standard security critical event set [4]. The SPC incorporates DLX as the execution engine with policies stored in its instruction memory. We implemented a representative debug infrastructure, based on a simplified version of ARM Coresight^{TM} features. It is functionally validated using Model-Sim [9] for typical use cases. Necessary interfaces/logic as described in Section 4, are added to the model to support DfD reuse for security policies. The DAP controls memory-mapped accesses to DfD instrumentation via the configuration bus. It also contains logic to control simultaneous debug and security requirements. Local DfD, similar to Coresight ETM/ITM are added for the functional IPs with their features enlisted in Table 2. Each has 16 32-bit configuration registers (64B address space) and support interfaces with the corresponding IP security wrapper. On security event detection, the configuration register based unique identifier (10 bits) is sent.

Table 3 summarizes the area and power overhead of the debug access port (DAP) to incorporate SPC access (which entails modification of DAP). The area estimation is provided from synthesis at 32nm predictive technology. Note that with respect to base DAP design, the area and power numbers are high because of the simple DAP logic in the model; however, the additional system overhead induced by

Table 3: Area (μm^2), Power (μW) of DAP (*SoC Area-* $\sim 1.42X10^6 \mu m^2$*; SoC Power-* $>$ *30 mW*)

DAP Area (Orig.)	New DAP Area	DAP Pwr (Orig.)	New DAP Pwr
380.2	527.67	12.63	19.82

the modification is negligible since the DAP contribution to overall SoC die area and power is minimal. The area and power overheads of DfD Trace Macrocells (TM) with respect to original TMs (without security wrapper interface) are enumerated in Table 4. The overheads are typically within 10%, but can be higher for some small IPs (*e.g.*, FFT, SPI).

Table 5 measures the decrease in hardware area overhead through re-purposing DfD for security wrappers. The measurement is done by comparing the current implementation with a reference implementation in which the security wrapper is responsible for detecting all necessary security-critical events, with no dependence on DfD. Note that the savings can be substantial, ranging from 20% to close to 60%. This is because a comprehensive DfD framework typically captures a majority of the security-critical events since they are also likely to be relevant for functional validation and debug.

Finally, Table 6 measures power overhead. This experiment is interesting since DfD reuse has two opposing effects: power consumption may increase since TMs remain active to collect security-critical events even when debug is not active; on the other hand, decreasing hardware overhead contributes to reduced power consumption. For DLX, the two effects cancel out, while for SPI and AES there is a net power overhead. Note that the overhead is minimal with respect to the overall power consumption of the entire SoC.

We end this section with an observation on interpretation of results. Note that the numbers provided are based on the security and DfD infrastructures in our SoC model; while reflective and inspired by industrial systems and policies, our implementations are much simpler. Nevertheless, we believe that the *overhead* measurements would substantially carry over to realistic usage scenarios. Perhaps more importantly, our experiments provide the guidance on parameters to analyze when considering re-purposing DfD for security implementation vis-a-vis standalone security wrappers.

7. RELATED WORK

Basak *et al.* [4] defines an architecture for security policies, but use dedicated security wrappers for event detection. Ray *et al.* [10] discuss trade-offs between security and

Table 4: Area (μm^2), Power (μW) Overhead of Modified DfD Trace Macrocells in SoC

DfD TM	Die Area (Orig.)	Area Ovrhd.(%)	Power (Orig.)	Power Ovrhd.(%)
DLX TM	15617	6.07	512	6.7
AES TM	5918	8.5	165	10.9
FFT TM	2070	18.8	60.6	20.2
SPI TM	2054.6	17.08	57.75	15.8
Mem. TM	4623	7.9	163.3	1.65

Table 2: Example DfD Instrumentation Features by IP Type in SoC Model

DfD by IP type	*Ex. DfD Inputs*	*Ex. Trigger Events*	*Ex. Trace Content*
DLX TM	Prog. counter (PC), Inst. opcode, Data RD/WR addr. (DA), Special register value, condition codes	PC in desired range, page, addr., Jump, Branch T/NT, Particular exception, interrupt, DA in specific page	Past, future 'N' inst. addr./values, status reg. values, next branch inst.
AES TM	Plain text(PT),Key,Cipher o/p(CT), Mode, Status, Intermediate round key	Encrypt/Dec. start/stop, Specific round reached, Key = desired	current 16B PT, Key, All future round keys, CT
SPI TM	Parameters, Status, i/p packet, Cycle counter, Configuration register	start/stop of operation, Source IP = desired, acknowledgement error	configuration register, past 'N' i/p packets
Memory Controller TM	Addr., Data, RD/WR, Burst size, Word/byte granularity, ECC	Addr in specific range, bank, DMA request, change in row buffer	In/Out Data word/byte, future 'N' data addr.

debug requirements in SoC designs. There have been research on exploiting DfD for protection against software attacks, *e.g.*, Backer *et al.* [11] analyzes the use of enhanced DfD infrastructure to confirm adherence of software execution to trusted model. Besides, Lee *et al.* [12] studies low bandwidth communication of external hardware with the processor via the core debug interface, to monitor information flow. Methods have also been proposed on securing SoC during debug/test [13].

8. CONCLUSION

We have developed a SoC security architecture that exploits on-chip DfD to implement security policies. It provides the advantage of flexibility and on-field update of security requirements, while being transparent to debug use cases. Our experiments suggest that the approach can provide significant benefit in hardware and area savings with no substantial energy overhead.

As part of future work, we plan to enable the architecture on complex, industrial SoC models and augment our architecture for more realistic security policies.

9. ACKNOWLEDGEMENTS

The work is funded in part by National Science Foundation grants 1603475, 1603480 and Semiconductor Research Corporation grant 2507.001.

10. REFERENCES

[1] J. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proc. 1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] M. Miettinen, S. Heuser, W. Kronz, A. Sadeghi, and N. Ashokan, "ConXsense: automated context classification for context-aware access control," in *ASIACCS*, 2014, pp. 293–304.

Table 5: Area (μm^2) Savings of IP Security Wrapper

Wrapper (corres. IP)	Orig. Area	New Area	Area Savings(%)
DLX μP	3437	2326	32.32
SPI cntrl.	1055	842	20.2
AES crypto	1661	702	57.7

Table 6: Power (mW) Analysis in SoC on implementation of Debug Reuse

IP	IP Power Consumption	Wrapper Savings	Corres. TM Power
DLX μP	6.54	0.52	0.551
SPI cntrl.	0.321	0.024	0.062
AES crypto	5.53	0.03	0.173

[3] M. Conti, B. Crispo, F. Fernandes, and Y. Zhauniarovich, "CRePE: A system for enforcing Fine-grained Context-related Policies on Android," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 5, pp. 1426–1438, 2012.

[4] A. Basak, S. Bhunia, and S. Ray, "A Flexible Architecture for Systematic Implementation of SoC Security Policies," in *IEEE ICCAD*, 2015.

[5] X. Wang, Y. Zheng, A. Basak, and S. Bhunia, "IIPS: Infrastructure IP for Secure SoC Design," *IEEE Transaction on Computers*, 2014.

[6] B. Vermueulen, "Design-for-Debug To Address Next-Generation SoC Debug Concerns ," in *IEEE ITC*, 2007.

[7] "CoreSight On-chip Trace & Debug Architecture, www.arm.com."

[8] S. Ray and Y. Jin, "Security Policy Enforcement in Modern SoC Designs," in *IEEE ICCAD*, 2015.

[9] "ModelSim - Leading Simulation and Debugging, www.mentor.com."

[10] S. Ray, J. Yang, A. Basak, and S. Bhunia, "Correctness and Security at Odds: Post-silicon Validation of Modern SoC Designs," in *ACM DAC*, 2015.

[11] J. Backer, D. Hely, and R. Karri, "On enhancing the debug architecture of a system-on-chip (SoC) to detect software attacks," in *IEEE DFTS*, 2015.

[12] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," in *ACM DAC*, 2015.

[13] J. Backer and R. Karri, "Secure Design-for-Debug for Systems-on-Chip," in *IEEE ITC*, 2015.