

System-on-Chip Security Architecture and CAD Framework for Hardware Patch

Atul Prasad Deb Nath¹, Sandip Ray², Abhishek Basak³ and Swarup Bhunia¹

¹Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA

²NXP Semiconductor, Austin, TX, USA

³Intel Corporation, Hillsboro, OR, USA

Abstract—System-on-Chip (SoC) security architectures targeted towards diverse applications including Internet of Things (IoT) and automotive systems enforce two critical design requirements: in-field configurability and low overhead. To simultaneously address these constraints, in this paper, we present a novel, flexible, and adaptable SoC security architecture that efficiently implements diverse security policies. The architecture and associated CAD flow enable “hardware patching” *i.e.* hardware security policy engine that can be seamlessly and securely upgraded in field to address unanticipated attacks or new security requirements. We implement (1) a centralized Reconfigurable Security Policy Engine (RSPE), (2) smart security wrappers, and (3) Design-for-Debug (DfD) infrastructure interface as the building blocks of the architecture. The proposed framework provides a systematic approach to represent and synthesize diverse security policies. Through extensive analysis using representative SoC models, we show, for the first time to our knowledge, that the proposed framework provides high level of patchability with minimal energy and performance overhead.

I. INTRODUCTION

With the rapid growth of Internet of Things (IoT) [1], the edge devices are given access to a tremendous amount of security assets everyday which must be protected from unauthorized or malicious access. The security assets can be broadly categorized into two groups: personalized end-user information (*e.g.*, contacts, location, browsing pattern, health information) and confidential collateral from architecture, design, and manufacturing, (*e.g.*, cryptographic and digital rights management (DRM) keys, fuses, debug instrumentation, defeature bits). With increased accessibility of IoT end-point devices to sensitive information, it is obligatory to protect these security assets from malicious and unauthorized access by developing security architectures and mechanisms in System-on-Chip (SoC) designs targeted towards emerging applications including IoT and automotive systems.

Over the past years, significant progress has been made on developing and implementing robust SoC security architectures [2], [4]. However, the application of these architectures are limited and it is quite difficult to adapt these in diverse IoT devices. In particular, most IoT applications impose two critical constraints: (1) in-field configurability to adapt to changing security requirements; and (2) operation under a tight boundary of energy and performance profiles. In this paper, we present a novel SoC security architecture designed to address the IoT application constraints, together with a complete methodology for implementing security policies. The proposed architecture includes the following three crit-

ical components: (1) a plug-and-play, flexible *infrastructure IP* called Reconfigurable Security Policy Engine (RSPE) to implement SoC security policies; (2) a standardized *security wrapper* architecture that enables communication of security-relevant events between various IPs in the SoC design and RSPE; and (3) a seamless interface between security wrappers and the on-chip SoC debug interface to enable requirements upgrade. The framework enables seamless and secure upgrade of security policies post-silicon and in-field, while addressing the performance and energy requirements. This enables, for the first time to our knowledge, an approach to developing “hardware patch”, *i.e.*, hardware implementation of security requirements that permit seamless post-silicon adaptation.

The paper makes three important contributions. First, we present a SoC security architecture, developed from the bottom up to enable hardware patching of security requirements. We show how to implement diverse SoC security policies including access control and information flow policies in the architecture, and how to adapt, modify, and upgrade these policies in field. Second, we present an automatic security policy mapping flow, providing a streamlined methodology for compiling high-level security policy definitions into the framework. Finally, we provide comprehensive quantitative analysis of energy, performance, and area overhead of the architecture realized with embedded FPGA. We demonstrate the efficacy of the architecture in terms of upgrading with large number of realistic security policies obtained through our industry contacts. We then compare our evaluation results with alternative implementations.

The remainder of the paper is organized as follows. Section II provides the relevant background on SoC security policies and our motivation for proposing the architecture. We discuss the proposed RSPE architecture in details in Section III. Section IV illustrates the CAD framework for automatically mapping and updating security policies in RSPE. Our experimental results are described in Section V in terms of major design parameters. We discuss related work in Section VI, and conclude in Section VII.

II. BACKGROUND AND MOTIVATION

A. SoC Security Policies

A significant component of SoC security is driven by the requirement to protect various on-chip assets against unauthorized access. Protection requirements to these security assets can be defined by confidentiality, integrity, and availability

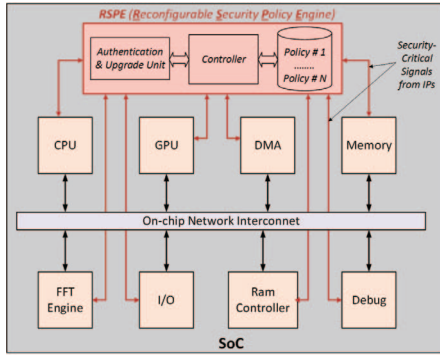


Fig. 1. An overview of a generic SoC architecture with the proposed RSPE (Reconfigurable Security Policy Engine).

properties [5]. The goal of a security policy is to map the requirements to “actionable” design constraints that can be used by SoC designers to develop protection mechanisms. Following are two representative examples of common security policy.

- *Example 1:* During boot, data transmitted by the crypto engine cannot be observed by any IP in the SoC fabric other than its intended target.
- *Example 2:* A secure key container can be updated for silicon validation but not after production.

Example 1 is a confidentiality requirement while Example 2 is an integrity constraint; The policies provide definitions of (computable) conditions to be satisfied by the design for accessing a security asset.

In addition to access control, security policies can capture requirements from information flow, liveness, time-of-check vs. time-of-use (TOCTOU), etc. A primary goal of SoC security architecture is to correctly and efficiently implement the security policies. Unfortunately, the state of the practice today depends heavily on human creativity. SoC designers use various architectural and system artifacts (*e.g.*, fuses, firmware, etc.) creatively to realize them in practice.

B. Motivation for In-field Hardware Patching

We consider two scenarios to illustrate the limitations of current practice of security policy implementation.

Case I: Attack on Confidentiality The attack scenario consists of two IPs, namely a trusted crypto engine IP A and an untrusted third party IP B. Ideally, the security policies to set the access control of different IPs are defined at the risk assessment phase. In practice, however, the policies go through continuous refinement through different phases of architecture. In some cases, the process gets extended to early design and implementation activities as new knowledge and constraints keep coming to light. Consequently, the security architects fail to implement a definitive information flow policy to map the IP-specific design constraints at the time of product launch *i.e.* time $t=0$. More importantly, the architects connect several IPs in same network-on-chip (NoC) in SoC with crypto block due to resource constraints and mark the IPs

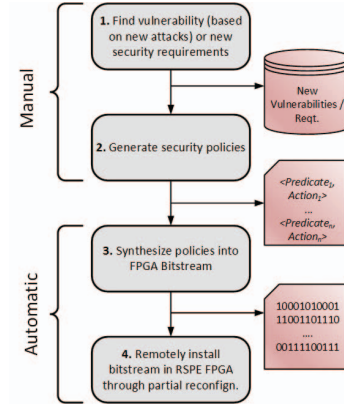


Fig. 2. Software flow for SoC security policy implementation in the proposed architecture.

as “safe” to observe part of the keys being exchanged in the communication fabric. Consequently, the untrusted IP B can be exploited by adversaries at a later time t in device life cycle to violate the information flow policy. For instance, the adversary can revoke *Key Obliviousness*, *i.e.*, exploit the malicious IP B to infer cryptographic keys by snooping data from crypto engine on the low-security communication fabric and gain illegal access breaching the confidentiality of the design [6]. The current mitigation approaches usually include the rigorous tasks of overhauling and upgrading the firmware.

Case II: Attack on Integrity An adversary can launch a code injection attack through a malicious or rogue IP by overwriting code segments via Direct Memory Access (DMA). For instance, the attacker can exploit the System Management Interrupt (SMI) handler to write to an address inside System Management RAM (SMRAM), which is basically part of DRAM reserved by BIOS SMI handlers. Based on the vulnerability, the adversary may have control over the address to write, the value being written, or both [7]. Preventive measures to thwart such attacks include identifying memory access requests to DMA-protected regions, and setting up mechanisms to bar DMA requests to all protected accesses.

In the aforementioned scenarios, the current approaches of mitigation fail to offer in-field adaptation of hardware as it involves multiple IPs and requires comprehensive changes in architecture and implementation. The proposed architecture overcomes these key limitations by enabling efficient and secure upgrade of policy implementations after deployment.

III. HARDWARE PATCHING INFRASTRUCTURE

Our key observation is that an SoC security policy can be viewed as a sequence of “commands” that specify how to react to certain behavior of IPs and inputs. These commands can be implemented in a separate, centralized IP which communicates with other IPs through a standardized interface. Based on this observation, we develop the following three architectural

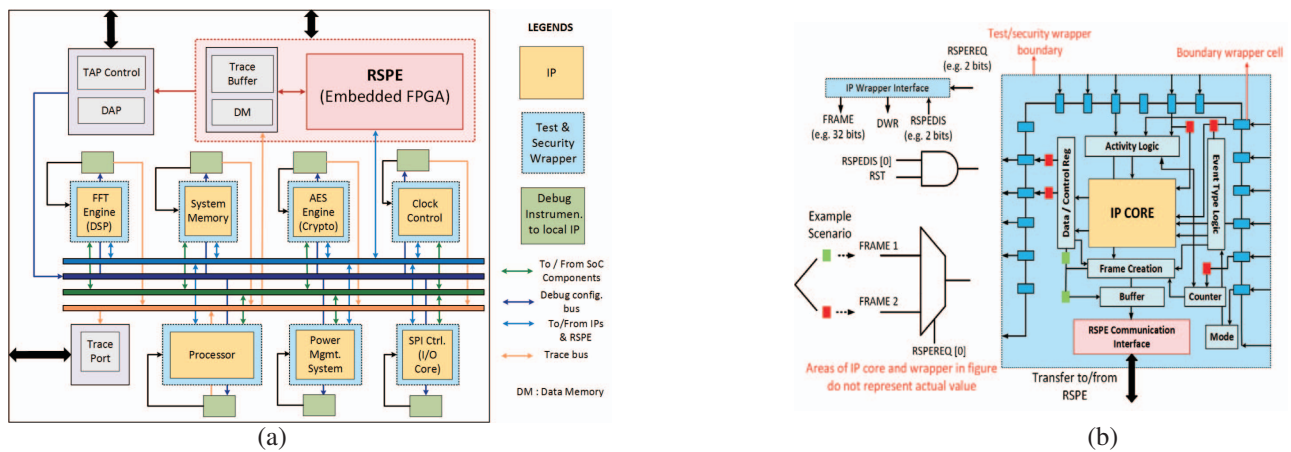


Fig. 3. Detailed illustration of the proposed security architecture. (a) RSPE acts as a centralized flexible security policy engine that enforces the security policies. (b) Security wrappers on individual IPs provide standardized, frame-based communication with RSPE.

Example Policy – IP B cannot read 1st 16 registers in address space of IP A, when A is doing a security critical computation

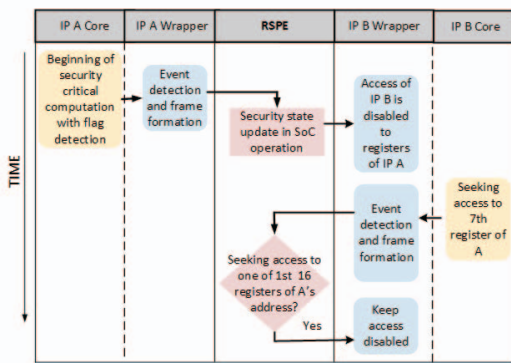


Fig. 4. Implementation of representative SoC security policy.

components to enable implementation of diverse policies in a way that is seamlessly configurable in field.

A. Reconfigurable Security Policy Engine (RSPE)

This block acts as the *security brain* of the SoC. It receives communication of relevant security events from the security wrappers in IPs, identifies the security state, and enforces mitigatory actions based on the enforced policies. The key enabler of in-field patching is the centralized location of all security policy implementations within RSPE. In particular, patching only requires updating the commands implemented in RSPE. This can be seamlessly performed if RSPE is in fact implemented on a reconfigurable hardware such as FPGA. The security wrappers do not need update even when the policies are changed in field. Since they are programmable at boot time by RSPE, an update only requires re-programming them to observe and control a potentially different set of signals that correspond to the updated policies.

B. Smart Security Wrappers

To implement security policies, RSPE must *know* some of the internal events of each IP and communications among IPs. This is achieved through the design of smart security wrappers, which essentially extend the test (*e.g.*, IEEE 1500 boundary scan based wrapper) and debug wrapper (*e.g.*, ARMs Coresight IP interface) which are already present for functional verification. The wrappers are programmable, so that they can be configured to monitor and control different sets of signals. RSPE configures the wrappers during boot for monitoring signals necessary to enforce the implemented security policies; during execution, RSPE identifies the security state of the system from the monitored signals, and if a policy violation is detected, RSPE can “allow” or “disallow” the event.

Fig. 4 illustrates an example policy implemented through this architecture. The policy prohibits access of first 16 (address-wise) internal registers of IP A by IP B when A is performing a security-critical computation. To enforce the policy, RSPE must know when B attempts to access particular local registers of A as well as the security state of the computation being performed by A at that instant. When IP A starts a security critical computation as indicated by a status flag, its security wrapper detects the event and communicates it with the RSPE. RSPE updates the security state of the SoC and disables accesses to all registers of A by B through control logic in B’s security wrapper. If B attempts to access a register bit (*e.g.*, register 7) in A’s address space to read a configuration value, the security wrapper of B detects this event of interest and informs the RSPE. The RSPE, determining that the request of B as a violation of the policy, denies corresponding access and maintains the disable status in B’s security wrapper.

C. Integration with Design-for-Debug (DfD) Infrastructure

To obtain controllability and observability over the required signals inside IP blocks, we interface RSPE with the on-chip Design-for-Debug (DfD) interface. A primary purpose of DfD is to provide access and control to signals that enable diagnosis of an unanticipated bug in field. Consequently, this

interface provides access to an extensive set of observable and controllable signals, which can be repurposed to realize security policies. We architect the 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 RSPE. To enable this functionality without hampering debug usages for the DfD, we implemented IP-level modification of the DfD logic including local debug instrumentation. In particular, noninterference with debug usage requires transmission of security data to RSPE via a separate port (instead of re-purposing debug trace port and bus), which incorporates additional trigger logic. To address this issue, we employed configuration register interface of corresponding DfD modules that can be configured by RSPE through trace port and then security-critical events can be logged into IP-specific security wrappers.

IV. PROPOSED CAD FRAMEWORK

A. Overall Flow and Major Steps

We have developed a CAD framework for synthesizing security policies into an RSPE implementation based on embedded FPGA. The framework has the following features: (1) it is amenable to automatic synthesis of arbitrary policies if the policies are described in specific predicate-action format, as shown in Table I, and when the required observable and controllable signals are accessible to the security policy engine; (2) it explores the design space to obtain energy-optimal policy implementations; (3) it allows incremental mapping of policies using partial reconfiguration for field upgrade; and (4) it integrates with the existing FPGA synthesis flow and exploits commercial application mapping tools.

Fig. 5 illustrates the major steps of the security policy mapping flow into a FPGA fabric. It integrates into conventional FPGA synthesis flow by adding two new steps (shaded in blue) in the front-end of the overall flow. In particular, it adds a pre-compilation stage where security policies are parsed and a register-transfer level description is created. The security policies are described as 3-tuple: **(timing, predicate, action)**. The timing information indicates either an operating mode or a timing information with respect to global clock. In case of policy 1 and 3 in Table I, the timing information is represented as the operational mode for the DLX processor (user mode) and the operational mode of whole SoC (debug mode), respectively. The predicate information indicates specific conditions based on the IP-internal observable signals or property of the interconnection fabric; the condition is expressed as a Boolean function of multiple observable signals. For the first policy, the predicate is formed by OR-ing two or more signals (e.g. Mem RD/WR Req. by User, Mem RD/WR Req. by IP A). The timing information is AND-ed with the predicate to create a combined condition. The third element is the action to be taken when the joint condition is true. This is done by asserting/deasserting signals or performing specific checks on a set of variables. For policy 1, the specific action is a check if the RD/WR address is within a range. The policies are parsed and

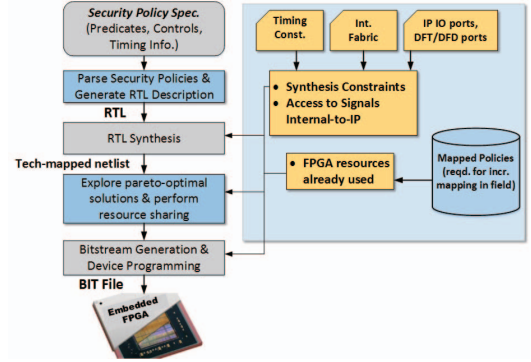


Fig. 5. Mapping diverse security policies on embedded FPGA-based RSPE.

an equivalent Verilog RTL code is generated by accounting for the information on IO ports for all IPs, test/debug ports, and interconnect fabric. The input/output is defined by considering all required observable and timing signals as input and all variables which are controlled as outputs. We represent each policy as an “assign” statement if it does not require using a state element, and as a separate “always” block otherwise.

The RTL module representing a set of policies is synthesized using FPGA synthesis tool. By considering the timing and other constraints, we generate a number of pareto-optimal mapping solutions. Resource sharing between several policies, in particular in the implementation of the predicate logic, is explored to optimize the area/energy requirement of the mapping under timing constraint. In case of incremental mapping of new set of policies on existing policies using the partial mapping solutions provided by the commercial FPGA mapping tools, we use the information on existing resource usage. From the pareto-optimal solutions, we select the best solution in terms of energy or area based on user preferences.

B. Authentication and Remote Upgrade

To protect the RSPE patch installation from attacks by malicious backdoors or Trojans, an authentication mechanism is employed based on secure challenge-response keys. SRAM Physically Unclonable Function (PUF) based keys are generated at power-up exploiting the intrinsic process variations. The reasons for choosing SRAM PUF include cost efficiency of weak PUF and avoidance of additional circuitry. The power-up key generation prevents any on-chip key storage access control attacks. The PUF-based authentication of the RSPE by a verifier at the update unit allows secure remote authentication and in-field patch installation.

V. RESULTS AND DISCUSSION

Due to lack of standard open-source SoC models, we developed an SoC design model that includes many major components of an industrial SoC design. It comprises of a 32-bit pipelined DLX microprocessor core (DLX), a representative memory controller IP, a 128b FFT engine, a 128b AES crypto core, a clock controller, a power management

TABLE I
REPRESENTATIVE SOC SECURITY POLICIES IMPLEMENTED ON THE PROPOSED ARCHITECTURE.

| Policy # | Predicate Part | Action Part | IPs Involved |
|----------|---|---|--|
| 1 | User mode & (Mem RD/WR Req. by User — Mem RD/WR Req. by IP A — ...) | RD/WR Addr. within specified range | DLX μP & any other IP with access to system memory |
| 2 | Supervisor mode & (Mem RD Req. by User — Mem RD/WR Req. by IP A — ...) | RD Addr. within shared memory range & No WR | DLX μP & any other IP with access to system memory |
| 3 | Debug mode & (Trace cells busy — power mgmt. module busy) | No update in power control firmware & no changes in SPI controller Config. Reg. | Power mgmt. module & SPI controller |
| 4 | !(Supervisor mode) & (Inst. Mem Update Req. through test access port or SPI controller) | No update of Inst. Mem. allowed | DLX μP |
| 5 | Active Crypto mode | No interrupt or Memory Access Req. from the DLX core or any IP is allowed | Crypto module, processor and other IPs access to processor |

TABLE II
ESTIMATION OF OBSERVABLE AND CONTROLLABLE SIGNALS

| IP Type | TW | SW | DfD | % Increment | |
|--------------|-------------|------|------|-------------|--------|
| Observable | DLX μP | 5 | 547 | 772 | 41.13 |
| | AES | 5 | 386 | 776 | 101.04 |
| | SPI | n/a | 104 | 161 | 54.81 |
| | Mem. | 5 | 224 | 608 | 171.43 |
| | FFT | 5 | 134 | 218 | 62.69 |
| Total | 20 | 1395 | 2535 | 81.72 | |
| Controllable | DLX μP | 1 | 142 | 255 | 79.58 |
| | AES | 1 | 107 | 188 | 75.70 |
| | SPI | n/a | 75 | 144 | 92.00 |
| | Mem. | 1 | 104 | 187 | 79.81 |
| | FFT | 1 | 76 | 156 | 105.26 |
| Total | 4 | 504 | 930 | 84.52 | |

TABLE III
ESTIMATION OF NUMBER OF SECURITY POLICIES

| Tuple Type | TW | SW | DfD | % Increment |
|------------|--------|-----------|------------|-------------|
| 2P, 1A | 570 | 490046760 | 2987015850 | 5.10 |
| 4P, 1A | 14535 | 7.91E+13 | 1.59E+15 | 19.16 |
| 8P, 1A | 377910 | 1.75E+23 | 3.89E+25 | 220.42 |
| 8P, 2A | 377910 | 4.42E+25 | 1.81E+28 | 407.94 |

unit, as well as a Serial Peripheral Interface (SPI) controller, all obtained from open-source IP databases from Opencores (<http://opencores.org>). A 32 KB central system memory IP is also integrated into it. To compare RSPE with microcontroller unit (MCU) based policy engine implementation, we augmented the SoC design with an alternative design described in previous work [6].

We implemented 10 representative security policies of varying complexity (cf. Table I), including different kinds of access control, as well as some instances of information flow, liveness, and secure boot. The entire framework has been functionally validated using ModelSim for typical policy use cases. All area, power and performance analysis are performed using 32nm CMOS technology library. We conducted our experiments on Cyclone V FPGA development board.

An estimation of the total number of observable and controllable signals with different design specifications, *i.e.*, *TW* (test wrappers), *SW* (security wrappers), and *DfD* (Design-for-Debug Infrastructure) is provided in Table II. The increase in observability and controllability over signals from various IP blocks is compared among the policy engines with test wrappers (*TW*), smart security wrappers (*SW*), and *DfD* interface (*DfD*). The column titled *% Increment* represents increase

in signal observability and controllability in *DfD* integrated design compared to security wrapper based policy engine.

Table III provides a comparative estimation of the number of arbitrary policies that can be implemented with *DfD* integrated RSPE and other designs. In Table III, *P*, and *A* represents *predicate* and *action*, respectively. We considered the observable signals from multiple IP blocks to determine the possible number of *predicates* and exploited the controllable signals to assert control and set the constraint through the *action* tuple of the policy framework. Table III shows the maximum limit on the number of security policies in several case scenarios *e.g.* 2 *predicates*, 1 *actions*; 4 *predicates*, 2 *actions*; 8 *predicates*, 2 *actions* etc. In every case, the *DfD* instrumentation results in a higher number of policies compared to other designs.

Table IV shows comparative overhead results for the FPGA implementation over the MCU version. Our testbench exercises 10 of these policies in succession in a specific scenario. For the MCU implementation, between 15 and 20 instructions are involved in the execution of one policy. The estimated dynamic and static powers are based on the signal activities in the representative test bench and standard voltage and thermal models at available 32 nm technology library. The total energy is calculated as the total power (dynamic + static) multiplied by the total latency incurred for the ten policies. For the FPGA implementation, the values of the parameters (die area, latency, total power, total energy) are calculated as follows. From the Altera Quartus tool, after compilation of the design, the reported number of arithmetic logic modules (both partially and fully utilized ones) is multiplied by the corresponding ALM area to obtain the total die area. The same testbench (as used in MCU version) is utilized to annotate signal activities for dynamic and static power calculation. To mimic the overhead of embedded FPGA, the total reported static power (reported for the whole FPGA chip) is multiplied by the total logic utilization factor to obtain the contribution towards the net leakage power. The FPGA implementation is around 5.02 X more energy-efficient than the MCU implementation. Furthermore, the MCU implementation takes on average 5.5 X more time compared to FPGA to execute these policies. The superior performance and energy efficiency are critical benefits of FPGA implementation since many IoT devices are energy-constrained and often require real-time security protection.

Table V provides a breakdown of energy consumption per policy. This is done by a testbench that executes each policy

TABLE IV
AREA, PERFORMANCE, POWER AND ENERGY VALUES FOR DLX PROCESSOR CORE AND EMBEDDED FPGA BASED SPE MODULE AND CALCULATED RATIOS FOR COMPARISON BETWEEN TWO APPROACHES.

| | Die Area (μm^2) | Clock Frequency (MHz) | Cycle Count (10 policies) | Total Latency (μs) | Dynamic Power (mW) | Static Power (mW) | Total Energy (nJ) |
|-------------|---------------------------|--------------------------|------------------------------|------------------------------|-----------------------|----------------------|----------------------|
| DLX μP | 0.724 | 203 | 210 | 1.04 | 14.27 | 63.48 | 80.86 |
| FPGA | 1.06 | 138 | 26 | 0.189 | 64.9 | 20.43 | 16.13 |
| Ratio | 0.68 | 1.47 | 8.07 | 5.49 | 0.22 | 3.11 | 5.02 |

TABLE V
RESULTS FOR EXECUTION OF EACH POLICY IN FPGA BASED RSPE

| Security Policy No. | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 | P10 |
|---------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| Energy (nJ) | 1.865 | 1.842 | 1.851 | 1.876 | 1.861 | 1.839 | 1.846 | 1.85 | 1.868 | 1.875 |
| Latency (ns) | 21.74 | 14.48 | 7.24 | 21.74 | 14.48 | 21.74 | 14.48 | 14.48 | 21.74 | 7.24 |
| Resources (ALMs) | 5465 | 4065 | 3260 | 5465 | 4065 | 4065 | 5465 | 4065 | 5465 | 3260 |

TABLE VI
COMPARISON OF AREA OVERHEAD FOR THE ENTIRE SoC

| SoC | Org. Area (μm^2) | μC Design Overhead (%) | FPGA Design Overhead (%) |
|----------------------------|----------------------------|--------------------------------|-----------------------------|
| SoC Model | 13.1×10^6 | 21.7 | 30.74 |
| Apple A6 (APL0598) | 96.71×10^6 | 2.92 | 4.26 |
| Qualcomm Snapdragon 800 | 118.3×10^6 | 2.39 | 3.49 |

in isolation. The energy costs across various policies are close because each policy represents some type of access control regulation and thereby involves similar computations on the individual IP frames. Besides, all of them incur 3 cycles (2 cycle read of corresponding event frame after buffer flag and 1 cycle execution to determine security state).

Table VI provides comparison of area overhead for the entire SoC between MCU and FPGA based implementation. Even though FPGA area is 0.68 X of MCU area, the total area overhead for realistic SoCs still less than 5%.

VI. RELATED WORK

Early research on security policies looked primarily on software systems and developed analysis frameworks for access control and information flow policies [8]. With modern SoC designs incorporating significant security assets, SoC security policies have become an area of significant research activities [9]. Basak *et al.* [6] defined an architecture for security policies using dedicated security wrappers. Ray *et al.* [10] discussed trade-offs between security and debug requirements in SoC designs. Backer *et al.* [11] analyzed the use of enhanced DfD infrastructure to confirm adherence of software execution to trusted model. Besides, Lee *et al.* [12] studied low-bandwidth communication of external hardware with the processor via the core debug interface, to monitor information flow.

VII. CONCLUSION

We have presented an architecture and a CAD framework for implementing SoC security policies that accounts

for the flexibility and in-field updates required by emerging applications. We have also presented an automatic synthesis framework that enables mapping arbitrary security policies into this framework and integrates with commercial tool flow. High adaptability and capability to implement diverse policies with minimal overhead are distinct features of the architecture. It simultaneously avoids the high performance and energy cost inherent in a software-based policy implementation. For a set of illustrative policies, we showed significant reduction in performance, power, and energy overhead.

ACKNOWLEDGMENTS

The work is supported in part by Semiconductor Research Corporation (SRC) grant 2649.001.

REFERENCES

- [1] D. Evans, "The internet of things - how the next evolution of the internet is changing everything," *White Paper. Cisco Internet Business Solutions Group (IBSG)*, 2011.
- [2] M. R. Sastry, I. T. Schoinas, and D. M. Cermak, "Method for enforcing resource access control in computer system," *US Patent 20120079590 A1*, 2012.
- [3] S. Krstic *et al.*, "Security of SoC Firmware Load Protocol," *HOST*, 2014.
- [4] ARM, "Building a secure system using trustzone technology," *ARM Limited*, 2009.
- [5] S. J. Greenwald, "Discussion Topic: What is the Old Security Paradigm," *Workshop on New Security Paradigms*, 1998, pp. 107–118.
- [6] A. Basak, S. Bhunia, and S. Ray, "A Flexible Architecture for Systematic Implementation of SoC Security Policies," *IEEE ICCAD*, 2015.
- [7] J. Loucaides and A. Furtak, "A New Class of Vulnerability in SMI Handlers of BIOS/UEFI Firmware," *The 15th Annual CanSecWest Conference (CanSecWest 2015)*, 2015.
- [8] J. Goguen and J. Meseguer, "Security Policies and Security Models," *IEEE Symp. on Security & Privacy*, 1982.
- [9] S. Ray and Y. Jin, "Security Policy Enforcement in Modern SoC Designs," *ICCAD*, 2015.
- [10] S. Ray, J. Yang, A. Basak, and S. Bhunia, "Correctness and Security at Odds: Post-silicon Validation of Modern SoC Designs," in *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," *DFTS*, 2015.
- [12] J. Lee, I. Heo, Y. Lee, and Y. Paek, "Efficient dynamic information flow tracking on a processor with core debug interface," *DAC*, 2015.
- [13] X. Li *et al.*, "Sapper: A Language for Hardware Level Security Policy Enforcement," *ASPLOS*, 2014.