ArtiFact: Architecture and CAD Flow for Efficient Formal Verification of SoC Security Policies

Atul Prasad Deb Nath, Swarup Bhunia and Sandip Ray
Department of Electrical and Computer Engineering
University of Florida, Gainesville, Florida 32608
Email: atulprasad@ufl.edu, swarup@ece.ufl.edu, sandip@ece.ufl.edu

***Abstract*— Verification of security policies represents one of the most critical, complex, and expensive steps of modern SoC design validation. SoC security policies are typically implemented as part of functional design flow, with a diverse set of protection mechanisms sprinkled across various IP blocks. An obvious upshot is that their verification requires comprehension and analysis of the entire system, representing a scalability bottleneck for verification tools. The scale and complexity of industrial SoC is far beyond the analysis capacity of state-of-the-art formal tools ; even simulation-based security verification is severely limited in effectiveness because of the need to exercise subtle corner-cases across the entire system. We address this challenge by developing a novel security architecture that accounts for verification needs from the ground up. Our framework, ArtiFact, provides an alternative architecture for security policy implementation that exploits a flexible, centralized, infrastructure IP and enables scalable, streamlined verification of these policies. With our architecture, verification of system-level security policies reduces to analysis of this single IP and its interfaces, enabling off-the-shelf formal tools to successfully verify these policies. We introduce a CAD flow that supports both formal and dynamic (simulation-based) verification, and is built on top of such off-the-shelf tools. Our approach reduces verification time by over 62X and bug detection time by 34X for illustrative policies.**

I. Introduction

Security assurance in System-on-Chip (SoC) designs is a highly critical area of research. Modern SoCs contain a variety of sensitive data (or "assets") that must be protected from unauthorized access. Such assets include private end-user information (*e.g.*, location, contacts etc.), cryptographic and DRM keys, proprietary firmware, and so on. Access to these assets and protection/mitigation requirements for unauthorized access attempts are governed by a collection of diverse *security policies*. It is of critical importance that the policies are implemented correctly. Indeed, a significant component of SoC security architecture entails developing techniques to enforce these policies ; correspondingly, a significant component of security verification involves ensuring that the design correctly executes the policies [1], [2].

Unfortunately, verification of security policies is non-trivial in current industrial practice. The hardware logics responsible for protection of various assets are implemented as part of system integration in conflation with various functional and optimization constraints, with little attention paid to ease of verification. These logics are sprinkled across different hardware and software blocks, collectively referred to as “IPs”. Consequently, *formal* verification of these logics requires discovery, analysis, and comprehension of system invariants that often span across the entire SoC design. Furthermore, even *dynamic* (simulation-based) verification is hard, since scenarios exercising security policies involve long, directed execution of corner cases in specific configurations, system execution, and environmental stimuli [3]. Validation of security policies is often left to complex *penetration testing* by human experts, and is typically incomplete. Unsurprisingly, security vulnerabilities are discovered in-field, often with disastrous consequences [4], [5].

In this paper, we take the position that a security architecture that facilitates formal verification needs is a feasible solution to the problem. To that end, we present an architecture (and a corresponding CAD flow) for security policy enforcement that is amenable to scalable formal verification of SoC security policies. We demonstrate via diverse realistic policy implementations that our approach can result in over 62X speed-up (on average) in formal policy verification using state-of-the-art commercial tools. Complex security policies in SoCs that could not be verified at all with traditional implementations become amenable for efficient verification using our framework. Furthermore, our work facilitates short counterexamples in presence of bugs in policy.

Our architecture includes a centralized *security policy engine*, a dedicated IP for implementing SoC security policies. Each policy is implemented within this IP as a state machine defined through a rigorous CAD flow. Policy enforcement entails communication of the policy engine with other IPs in the SoC : this is performed by a standardized protocol. Centralized policy implementations have an important characteristic to facilitate scalable formal verification : a proof of correctness is typically confined to the policy engine and its interfaces and is oblivious to design invariants in any other IP blocks. Note that all other conditions remaining equal, complexity of formal verification is typically proportional to the size of the design block being analyzed [6]. Consequently, by enabling the target of formal analysis to be confined to a single IP, we enable significant scalability in verification of security policies over traditional distributed implementation.

Our work builds upon and extends previous work [7]–[9] on systematic, flexible architecture for SoC security policy implementations. Analogous to these works, our approach involves a centralized policy implementation framework. However, all previous works were focused on flexibility of implementations ; verification was not considered. We show

2159-3477/18/$31.00 ©2018 IEEE
DOI 10.1109/ISVLSI.2018.00081

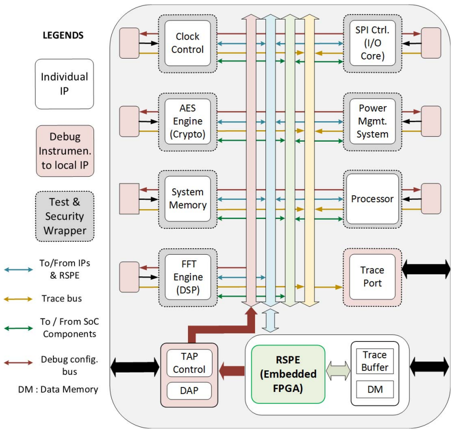

Fig. 1: Proposed Security Architecture : RSPE acts as a centralized flexible security policy engine to enforce policies.

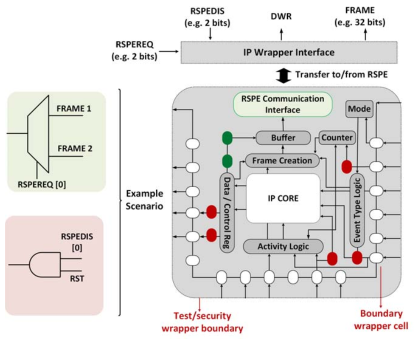

Fig. 2: Wrapper Architecture : security wrappers on individual IPs provide standardized, frame-based communication.

how to extend such frameworks with augmented CAD flows for effective verification methodology of security policies.

The paper makes three major contributions. First, we propose, for the first time to our knowledge, a *formal security verification flow* for security policies that directly influences and exploits architectural support for policy enforcement. Architectural support reduces the complex problem of formally verifying policy implementations to a simpler task of analyzing a single infrastructure IP, thereby providing verification scalability. Second, we demonstrate the efficacy of the framework in identifying and root-causing bugs. Our framework reduces root-causing efforts by providing short counterexamples enclosed to a single IP. This facilitates streamlining the debug flow significantly compared to directed testing and fuzzing approaches used in current industrial practice. Finally, we develop a comprehensive evaluation of both formal and dynamic aspects of verification on a wide diversity of realistic security policies implemented on an illustrative SoC model. The policies we consider include IP-specific as well as system-level security constraints.

II. Background

A. SoC Security Policies

The goal of an SoC security policy is to map the security requirements to design constraints to develop protection mechanisms. Following are two representative examples :

- *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.

The policies may vary depending on the state of execution (*e.g.*, boot time, normal execution), or position in the development life-cycle (*e.g.*, manufacturing, production). In addition to access control, security policies can capture requirements from information flow, liveness, etc.

B. SoC Security Architecture

Our work builds on a centralized SoC security architecture introduced in previous work [7]. It includes the following :

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 implemented policies.

Smart Security Wrappers. The idea for security wrappers is to enable IPs to communicate security-critical events to RSPE. The wrappers are programmable, so that they can be configured to monitor and control different sets of signals. RSPE configures the wrappers during boot time for monitoring signals necessary to enforce the security policies.

Interface with Design-for-Debug. RSPE is interfaced with the on-chip Design-for-Debug (DfD) interface. This interface provides access to an extensive set of observable and controllable signals inside IPs, which can be exploited for verification and re-purposed for realizing new policies.

III. Problem Analysis

A. Existing Challenges

Security policies in current practice are implemented by starting with a baseline architecture which is iteratively refined as follows :

- Use threat modeling to identify potential threats to the current architecture definition.
- Refine the architecture with mitigation strategies covering the threats identified.

The baseline architecture is derived from legacy SoC designs. For each asset, the architect must identify (1) who can access the asset, (2) what kind of access is permitted, and (3) at what points in the system life-cycle such access requests can be granted. The current industrial practice for verifying policies includes functional tests, fuzzing tests, and penetration tests. The scale of formal tools is limited to policies involving only one or a few IPs [3].

B. The Need for Architectural Support

Despite being an integral part of system development flow, the verification requirements of modern industrial designs are barely met by state of the art technologies. Formal methods provide an effective paradigm for security policy validation since they can provide a mathematical guarantee

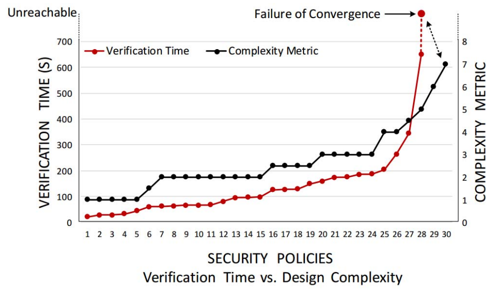

Fig. 3: An illustrative example of required verification effort with increasing design complexity and an eventual failure of traditional formal verification approaches.

of correctness of the policy enforcement which is unavailable from dynamic (fuzzing, functional simulation, and directed testing) techniques. However, it is imperative to develop architectural support that can make formal analysis of realistic security policies scalable : it is crucial for architectural support that ensures that invariants needed can be enclosed within a small block of logic. Our work facilitates this by basing our CAD flow on top of a centralized framework.

Furthermore, it is important to study the role of policy complexity on verification time. We developed a new metric of policy complexity based on empirical analysis. Fig. 3 shows the correlation between actual verification time and the proposed metric. The *complexity metric* C_m is defined as follows :

$$C_m = R_c + \sum_{i=1}^{n} \frac{E_i}{2}\left[\frac{O_i}{S_i} + \frac{C_i}{S_i}\right] \tag{1}$$

Here, S_i is the number of security critical signal(s) involved in a policy, E_i is the number of security event(s) triggered by the policy, O_i and C_i is the number of observable signal(s) and controllable signal(s) of IPs involved in the corresponding policy, respectively. An additional complexity constant R_c is introduced by the RSPE in case of the proposed architecture. The soaring verification time (Fig. 3) with increasing complexity of policies signifies the limitation of conventional architecture to scale with verification needs.

IV. Proposed Framework

A. Architectural Support

Our architecture builds on the centralized policy engine developed in previous work [7]. Transforming an architecture primarily developed for policy *implementation* (without verification concerns) to support effective formal verification is non-trivial. Here we list some architectural modifications that are crucial to the verification need.

Event Logging in RSPE : To facilitate the optimum event detection via centralized architecture, we supplement RSPE with augmented of event logging capability. The improved event logging is enabled by incorporating configuration register and special purpose registers for storing event trigger, transfer, and related meta data based on event type and requirement. The increment in security events logged by RSPE reduces the complexity in security policy verification process by minimizing the reachable trace lengths for verification and violation paths. The centralized implementation of policies and corresponding security properties facilitates the tool to access the required signals in a shorter period of time.

Event Repository in Local DfD : A key criteria for the implementation and verification of arbitrary security policies of varying complexity is the detection of large number of security critical events in the SoC. System level security polices of higher complexity often require user-defined triggers and custom interrupts in inter-IP communications. We developed an augmented repository of security critical events by exploiting the configuration registers in local debug instrumentations. We used on-chip local debug modules with configuration registers and associated logic to map an extended number of security events for system level policies.

Enhanced Interconnect Fabric : We augmented the interconnect fabric of our SoC model for inter-IP communication by establishing a shared memory bus. To facilitate system level interaction between IPs, we mapped the control registers for each IP to specific addresses of system memory range and utilized the corresponding control signal interfaces to respond to incoming transactions from other functional IPs. In case of incoming interrupts and requests during active computation mode of an IP, a disable signal is instantiated by the policy engine for triggered events. For instance, all request and interrupts from rest of the IPs are invalidated when AES engine is in crypto mode. Consequently, any unauthorized access requests during crypto mode is logged into configuration registers as potential attempts of violation.

B. CAD Flow

We automatically synthesize policies into RSPE-based architecture. The policies are parsed as *action-predicate* tuples. The principle of pareto-optimality is employed in the synthesis procedure for energy optimum implementation. Fig. 4 illustrates the design flow. The flow introduces a pre-compilation stage, where security policies are parsed and a register-transfer level description is created for a control state machine that implements the action-predicate tuple ; this is integrated with an FPGA synthesis flow to create a reconfigurable policy implementation.

Given the above flow, formal property synthesis entails designing of a *monitor state machine* C_p for each policy P. The goal of the monitor state machine is to “watch” C_p and output 1 if p ever makes a deviation from its expected transitions and 0 otherwise. The formal property then reduces to the assertion that C_p never outputs 1. Note that in addition to its use in formal verification, C_p can also act as a runtime monitor for the assertion p : this is relevant in case of a policy p that cannot be completely verified (*e.g.*, if the correctness entails hardware/software co-execution and cannot be established from the hardware alone). In practice, we define C_p by augmenting the RTL design and used primarily for formal verification. If the verification succeeds the monitor C_p is no longer necessary, and can be safely removed : however, if the verification fails or is inconclusive,

TABLE I: System Level Security Policies Implemented on the Proposed Architecture.

Policy #	Predicate Tuple	Action Tuple	Corresponding IPs
1 : Read / Write operation of IPs within system memory range in user mode	(Mode : User) & (Memory read/write request by user or any other IP)	Read/write address within specified range	Any IP with access to system memory
2 : Read / Write operation of DLX uP to shared memory range in shared memory range	(Mode : Supervisor) & (Memory read/ write request by user or any other IP)	Read/write address within shared memory range & No write	Any IP with access to system memory
3 : Interrupts (e.g. reset, immediate result, change of key etc.) from all IPs are prohibited during active crypto mode	(Mode : Active crypto) & (Access request by user or any other IP)	No interrupt or memory access request from any IP is allowed	Crypto module and any other IP with access to crypto core
4 : Read / Write access of IPs to round key registers are prohibited during active crypto mode	(Mode : Active crypto) & (Read/write request to round key registers by any IP)	No read/write access to round key registers by any IP is allowed	Crypto module and any other IP with access to crypto core
5 : Interrupts by Power management module (clock freq. change, reset, enable, go) during active computation	(Mode : Active computation) & (Interrupt or access request from PMC module)	No interrupt or access request from PMC module is allowed	PMC module and any other IPs accessible to PMC
6 : All IPs' access to interconnect fabric is prohibited during crypto key transfer	(Mode : key in transfer) & (Interrupt or access request by any IP)	No interrupt or access request from IPs to interconnect fabric is allowed	All IPs with privilege to access interconnect fabric
7 : Interrupts from all IPs are prohibited during μP core instruction memory update	(Mode : Supervisor) & (μP instruction memory update) & (Access requests)	No access request from any IP is allowed	μP core and any other IP

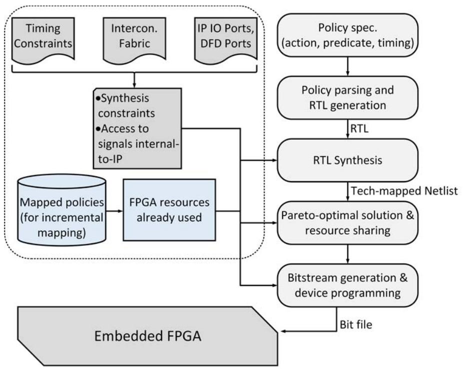
Timing Constraints
Intercon. Fabric
IP IO Ports, DFD Ports
•Synthesis constraints
•Access to signals internal-to-IP
Mapped policies (for incremental mapping)
FPGA resources already used
Policy spec. (action, predicate, timing)
Policy parsing and RTL generation
RTL
RTL Synthesis
Tech-mapped Netlist
Pareto-optimal solution & resource sharing
Bitstream generation & device programming
Bit file
Embedded FPGA

Fig. 4: CAD Flow for Mapping Security Policies on RSPE.

we keep the augmented RTL and connect the output of C_p to additional routines that can perform mitigatory action if the failure occurs runtime.

The steps for property mapping is quite straightforward : C_P can be synthesized mechanically from the state machine of P, and is well-established in current industrial practice. However, without centralized RSPE, there would be no systematic way for writing these assertions in a traditional SoC design. This further outlines the critical role of architectural support for security verification.

V. Results

A. Experimental Setup

The SoC model includes a 32-bit pipelined DLX microprocessor core (DLX), a 32 KB central system memory, a standard memory controller IP, a 128b AES crypto core, a 128b FFT engine, a clock controller, a Serial Peripheral Interface (SPI) controller, and a power management unit. The IPs were obtained from Opencores (`http://opencores.org`). The security policies are mapped on an embedded FPGA-based RSPE that act as the execution engine. For experiments, we developed two versions of the SoC model, a baseline design and RSPE-based design. In the baseline SoC, each IP is augmented with standard boundary scan based wrappers (i.e. IEEE 1500) for detection of local events. Security policies in baseline model are implemented over the constituent IP cores in a distributed manner. In the RSPE-integrated model, we enhanced the IPs with smart security wrappers, and developed interface for *DfD* integration.

B. Formal Verification Results

Our formal verification results use off-the-shelf tool JasperGold [10]. We synthesized assertions in RSPE as discussed in Section IV-B. To compare efficacy, we implemented the same policies on a baseline SoC, paying specific attention to traditional performance optimization to reflect the current state of practice ; assertions were also developed for this model. An Intel®Core^{TM} i5-3427U CPU (1.8 GHz) with 8 GB memory is used to run verification on a linux server.

System-level Security Policies. We implemented 7 (P1 to P7) system-level security policies of varying complexity in our SoC model (cf. Table I). Table II summarizes the verification results for the system level policies implemented in base-line and RSPE-based design. All proven assertions in the table are "Infinite" bound type meaning the proofs are exhaustive and expected to hold true under all circumstances. We chose multi-engine environment with multi-property settings for optimum exploitation of the tool.

The increase in verification time with policy complexity is evident in the results presented in Table II. For instance, the interrupt handling policy (P#3) of AES during crypto mode require initiation of all possible incoming transactions coming from each of the IPs. The higher verification time, in this case, for DLX interrupts can be attributed to an increased number of security event association. Note that for illustrative system level policies, our approach reduces verification time by a maximum of 62X compared to baseline implementation.

IP Specific Security Policies : We implemented 8 representative IP-specific policies. Table III summarizes the verification report provided by the tool for baseline and proposed design. The verification of IP specific policies requires less effort in both baseline and RSPE based design due to the ease of observing and controlling involved signals, with corresponding low verification time.

C. Scalability Analysis

To demonstrate the scalability of our approach, we consider a case study of boot integrity check policies. These policies verify the trustworthiness of the system at power-on. The implementation of these policies in our model mandates checks for AES crypto engine's data path along with system boot processes (including power-on-self-tests, firmware

TABLE II: Proof of Correctness Results : System Level Security Policies

Formal Verification Results (Baseline vs RSPE)										
Security Policies	IP Cores Involved	Baseline				RSPE				Reduction (times)
		JG Engine Mode	Proof Effort	Bound	Avg. Time (s)	JG Engine Mode	Proof Effort	Bound	Avg. Time (s)	
P #1	DLX up, AES, FFT, SPI	N	1-2	Infinite	27.246	Hp	1	Infinite	1.1435	23.827
P #2	AES, FFT, SPI, PMC	Ht	1	Infinite	64.429	N	1	Infinite	1.387	46.452
P #3	AES, FFT, SPI, PMC	Bm, Hp. I	1-13	Infinite	218.97	Ht, Hp, I	1-4	Infinite	10.824	20.23
P #4	DLX uP, PMC, FFT, SPI	Bm, Hp, Ht,	1-7	Infinite	126.2	N	1-3	Infinite	2.03	62.168
P #5	DLX Up, PMC, FFT, SPI	Bm, D, I,Hp	1-11	Infinite	303.3	I, U, Hp, Ht	2-7	Infinite	21.112	14.366
P #6	DLX uP, PMC, FFT, SPI	Hp, Ht,N	1-3	Infinite	62.958	Ht	1	Infinite	2.7868	22.592
P #7	AES, FFT, SPI, PMC	D, Bm, Hp,N	1	Infinite	142.9	N, Ht	1	Infinite	6.9305	20.619

TABLE III: Proof of Correctness Results : IP Specific Security Policies

Comparative Results for IP specific Security Policy Implementation (Baseline vs RSPE)						
Security Policies for DLX Up	Baseline			RSPE		
	JG Engine Mode	Bound	Time (s)	JG Engine Mode	Bound	Time (s)
DLX core instruction memory can only be updated at supervisor mode	Ht	Infinite	1.589	Hp	Infinite	0.695
DLX mode of operation cannot be unknown at startup	N	Infinite	0.615	N	Infinite	0.714
DLX power (high/low) mode of operation check at startup	N	Infinite	0.519	Ht	Infinite	0.936
External read write to DLX to only I/O mapped data memory region	N	Infinite	1.698	Ht	Infinite	1.796
Security Policies for AES Crypto Core	Baseline			RSPE		
	JG Engine Mode	Bound	Time (s)	JG Engine Mode	Bound	Time (s)
AES key cannot be unknown at startup	Hp	Infinite	0.875	Hp	Infinite	1.537
Key check against previous set (nonce/to prevent replay attack)	Hp	Infinite	1.328	N	Infinite	2.914
In crypto mode, cipher text output interface is disabled	Ht	Infinite	1.537	Hp	Infinite	1.271
AES power (high/low) mode of operation check at startup	Ht	Infinite	0.505	Hp	Infinite	0.713

TABLE IV: Results on Scalability Analysis on Baseline and RSPE-based Design

Use Case Scenario : Comprehesive Policy Implementation									
	IP Cores Involved	Baseline				RSPE			
		JG Engine Mode	Result	Bound	Time (s)	JG Engine Mode	Result	Bound	Time (s)
Policies : Boot Integrity Check	DLX uP,	Ht	Undetermined	1021	31056.2	Ht	Proven	Infinite	12984.8
	AES, SPI,	D	Undetermined	872	9246.8	D	Proven	Infinite	4298.8
	FFT, PMC,	I	Undetermined	1543	22898.9	I	Proven	Infinite	15998.6
	Sys memory	Hp	Undetermined	987	18449.7	Hp	Proven	Infinite	11365.3

TABLE V: Results on Bug Detection in Security Policies for Baseline and RSPE-based Design

Funtional Verification Results (Baseline vs RSPE)						
Security Policies	IP Cores Involved	Baseline		RSPE		Reduction (times)
		Detected Bugs	Time (s)	Detected Bugs	Time (s)	
P #1 - P#7	DLX Up, AES, FFT, SPI, PMC	N/A	>86400*	N/A	>86400*	N/A

Footnote : *The simulation run time was greater than 24 hours i.e. >86400s. No bugs were detected within the time limit.

Formal Verification Results (Baseline vs RSPE)										
Security Policies	IP Cores Involved	Baseline				RSPE				Reduction (times)
		JG Engine Mode	Proof Effort	Bound	Max Time (s)	JG Engine Mode	Proof Effort	Bound	Max Time (s)	
P #1	DLX up, AES, FFT, SPI	N	1-2	Infinite	32.675	Hp	1	Infinite	0.613	34.351
P #2	AES, FFT, SPI, PMC	Ht	1	Infinite	67.145	N	1	Infinite	1.573	15.686
P #3	AES, FFT, SPI, PMC	Bm, Hp. I	1-13	Infinite	345.489	Ht, Hp, I	1-4	Infinite	6.954	26.651
P #4	DLX uP, PMC, FFT, SPI	Bm, Hp, Ht,	1-7	Infinite	186.228	N	1-3	Infinite	1.915	30.155
P #5	DLX Up, PMC, FFT, SPI	Bm, D, I,Hp	1-11	Infinite	263.746	I, U, Hp, Ht	2-7	Infinite	13.573	14.602
P #6	DLX uP, PMC, FFT, SPI	Hp, Ht,N	1-3	Infinite	80.174	Ht	1	Infinite	1.688	25.973
P #7	AES, FFT, SPI, PMC	D, Bm, Hp,N	1	Infinite	185.259	N, Ht	1	Infinite	7.436	13.731

integrity check, and peripheral core integrity check). Table IV summarizes the verification report. For the baseline design, the verification engines of the tool failed to reach convergence for integrity check policies with multiple engine modes. However, the formal proof is completed in the RSPE-based implementation where the state space explosion phenomenon is avoided through the centralized implementation. This suggests that with architectural support, it is possible to address scalability limitations in verification and potentially formally verify complex system-level security policies.

D. Bug Detection

To evaluate the robustness of RSPE in bug detection, we injected a set of bugs in the system-level policies. The bugs were inserted with close interaction with industry and are representative of real security bugs detected in industrial environments. Furthermore, the selection aims to cover the spectrum of confidentiality, integrity, and availability requirements of assets in modern SoCs.

Access to Memory Bug. We considered a violation scenario where the state machine controlling the memory address register fails to detect the address range breach in the cur-

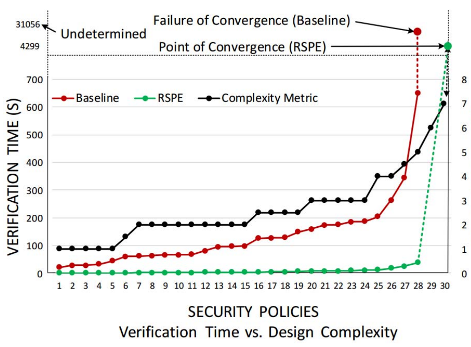

Fig. 5: An illustrative example of incremental verification effort and scalability of RSPE with design complexity.

rent/overlapping clock cycle. The bug, if goes unmitigated, can lead to unauthorized access of a malicious attacker or restricted IP to secure memory address range. The possible consequences of such breach include a violation of confidentiality and integrity of the assets of secure memory.

Active Crypto Mode Bug. In this violation scenario, the status of active crypto signal remains asserted throughout the crypto sequences and is not de-asserted once the operations are finished. The bug can hamper the secure flow of operation as the IPs are blocked from accessing the crypto assets after a crypto operation. The event leads to violation of availability property and consequent unavailability of assets.

Active Computation Mode Bug. We considered a violation scenario where the state machine controlling mode of operation of IPs gets stuck in the current state leading to functional failure. The bug is representative of the functional failure of the SoC due to a loss of availability. It directly affects the incoming transactions from power management module and causes stagnation in the flow of execution.

Results Analysis. We tested the policies with bugs in a simulation environment. The summary of *functional verification results* is illustrated in Table V. We employed constrained random testing via ModelSim for assertion based dynamic verification. Random functional testing failed to detect any of the bugs in reasonable time (>24 hours), which highlights the limitation of traditional security policy verification approaches in SoC designs. Table V also shows the summary of *formal verification results* for 7 system level security policies. The trace lengths of counterexamples in baseline design are significantly higher than the trace lengths of RSPE-based design. With RSPE, the engines of formal tool are able to find violation traces with minimal trace attempts leading to reduced trace lengths of counterexamples and improved verification time. Our approach reduces counter example detection time up to 34X compared to base-line design.

VI. Related Work

Several formal methods have been proposed for the verification of security properties [11]–[13].The focus of these works are hardware security issues i.e. malicious hardware Trojans, side-channel attacks, etc. Their application, however, is limited by the failure to scale with design complexity. Though novel techniques have been proposed for improvement [14], state space explosion is still the major limitation of proving security properties in large SoC designs. Research efforts have been made to address SoC security and verification issues by developing scalable architectural frameworks. Infrastructure IPs are employed to facilitate SoC functional verification, testing, and yield improvement [2], [15]. However, these approaches lack scalable architectural features like centralized or flexible infrastructure IP, standardized interface with IP blocks, and systematic CAD flow.

VII. Conclusion

We have developed an architectural framework for efficient and scalable formal verification of complex security policies on SoC platforms. Our work, for the first time to our knowledge, marries two highly crucial but typically isolated components of security assurance, architecture and validation. We show how to develop an architecture that not only enables systematic policy implementation but also scalable analysis and formal verification. The experimental results on realistic SoC models and policies suggest that the approach can reduce verification time for system-level policies by orders of magnitude, help verification of arbitrary policies with varying complexity, and significantly aid the detection of bugs deeply rooted inside the design. Future work will involve enabling the architecture on industrial SoC models and silicon verification.

References

[1] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of SoC Firmware Load Protocol," in *HOST*, 2014.

[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. Ray, E. Peeters, M. Tehranipoor, and S. Bhunia, "System-on-Chip Platform Security Assurance : Architecture and Validation," *Proceedings of the IEEE*, 2018.

[4] Homebrew Development Wiki, "JTAG-Hack," http://dev360.wikia.com/wiki/JTAG-Hack.

[5] L. Greenemeier, "iPhone Hacks Annoy AT&T but Are Unlikely to Bruise Apple," *Scientific American*, 2007.

[6] R. Kaivola, S. Pandav, A. Slobodova, C. Taylor, V. A. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation," in *CAV*, 2017.

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

[8] A. P. D. Nath, S. Ray, A. Basak, and S. Bhunia, "An Architecture and CAD Flow for Hardware Patch," in *ASPDAC*, 2017.

[9] A. Basak, S. Bhunia, and S. Ray, "Exploiting Design-for-Debug for Flexible SoC Security Architecture," in *DAC*, 2016.

[10] "JasperGold : Formal Property Verification App," 2017, www.jasper-da.com/products.

[11] S. Drzevitzky, "Proof-carrying hardware : Runtime formal verification for secure dynamic reconfiguration," in *FPL*, 2010.

[12] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *VTS*, 2012.

[13] M. Rathmair and F. Schupfer, "Hardware trojan detection by specifying malicious circuit properties," in *ICEIEC*, 2013.

[14] X. Guo, R. G. Dutta, P. Mishra, and Y. Jin, "Scalable SoC Trust Verification using integrated theorem proving and model checking," in *HOST*, 2016.

[15] Y. Zorian, "Embedded memory test and repair : Infrastructure IP for SoC yield," in *ITC*, 2002.