# DiSPEL: A Framework for SoC Security Policy Synthesis and Distributed Enforcement

Sudipta Paria, Aritra Dasgupta, Swarup Bhunia
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL, USA

2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-7394-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/HOST55342.2024.10545407

***Abstract*—Modern System-on-Chip (SoC) designs that rely on bus architectures are susceptible to a range of hardware and software threats, necessitating the implementation of diverse security measures. The protection of valuable assets against unauthorized access requires the integration of various security policies. The complex interactions among multiple Intellectual Property (IP) blocks within an SoC pose significant challenges for stakeholders such as SoC designers, system validators, and security experts, who must discern relevant policies, implement them, and ensure compliance. The manual expertise needed for upgrading policies and adapting IPs to meet varying security requirements significantly impacts both design costs and time-to-market. This paper introduces DiSPEL, a flexible and efficient framework designed to automatically synthesize and enforce security policies expressed in a simple grammar format for any bus-based SoC design. DiSPEL adopts a distributed deployment strategy to maintain the integrity of trusted bus operations, even when dealing with untrusted IPs. DiSPEL achieves policy enforcement by (i) incorporating a dedicated centralized module to address bus-level security specifications and (ii) generating the necessary logic and appending it to the IP-level bus wrapper to meet IP-specific requirements. The proposed framework supports generic security policy types, accommodating both synthesizable and non-synthesizable constructs. Experimental results validate the efficacy and correctness of DiSPEL in enforcing security requirements, demonstrating its practicality with minimal overhead in terms of area, delay, and power consumption. These results are based on experiments conducted using open-source standard SoC benchmarks.**

***Index Terms*—SoC Security, Security Policies, Formal Representation, Vulnerabilities, Threats, Design-for-Security**

## I. Introduction

Computing systems in modern times are commonly developed as System-on-Chip (SoC) designs, seamlessly integrating numerous functionalities into a single integrated circuit. These SoC configurations consist of diverse design modules, referred to as Intellectual Properties (IPs), which collaborate via on-chip communication fabrics to achieve the intended system functionality. In a typical SoC design, secure assets are distributed among various IPs, necessitating protection from unauthorized access. These assets encompass critical components like cryptographic keys, Digital Rights Management (DRM) keys, programmable fuses, on-chip debug instrumentation, defeature bits, and more. Effectively safeguarding the secure assets of an SoC from potential threats requires the incorporation of design-time considerations. This proactive approach is crucial for preventing potential attacks and facilitating detection and recovery in the event of an attack. Secure assets in modern SoC designs are dispersed across multiple IPs, demanding protection against unauthorized access that could result in severe consequences, such as identity theft, leakage of sensitive data, and significant financial losses. Ensuring the protection of these secure assets necessitates the essential implementation of security policies. These policies meticulously define the authentication, access, and protection requirements for diverse assets within the design. Generally, these policies reflect confidentiality, integrity, and availability requirements at a high level, thus providing actionable guidance to SoC designers and architects, ensuring the implementation of suitable protection strategies. Efficient implementation of these policies and validation to ensure effective policy enforcement is a critical step in an SoC integration process. These steps, however, have become increasingly challenging due to complex policies involving a collection of IPs and assets and protection requirements across various classes of potential adversaries. Typical policy enforcement starts with a baseline architecture, which is refined over multiple iterations of the following steps:

- Employ threat modeling to recognize potential threats.
- Integrate mitigation strategies to address threats identified.

The responsibility of security architects/experts includes identifying three crucial elements for each asset: (i) authorized users, (ii) permissible access types, and (iii) specific approval or denial stages during system execution. However, this procedure can be exceedingly intricate and time-consuming, mainly because of the significant number of assets present in a typical SoC design. These assets can either have static definitions or be dynamically generated at different IPs during the execution of the system, which adds to the overall complexity. As SoC designs become increasingly intricate, integrating Design-for-Security (DfS) mechanisms has become a significant challenge. The main obstacles are as follows: (1) addressing threats at the architecture level along with IP level issues, (2) incurring acceptable additional overhead while upholding the functional correctness of IPs, (3) avoiding design or test conflicts due to multiple DfS approaches, (4) achieving comprehensive protection when using insecure third-party IPs.

The existing solutions, such as IIPS [1], E-IIPS [2], RSPE [3], and MSIPS [4] offer policy enforcement through smart wrappers and infrastructure IPs in bus-based SoC primarily handling the architecture-level or IP core-level threats separately at runtime. However, they do not cater to diverse customizable security needs as defined by the user using a high-

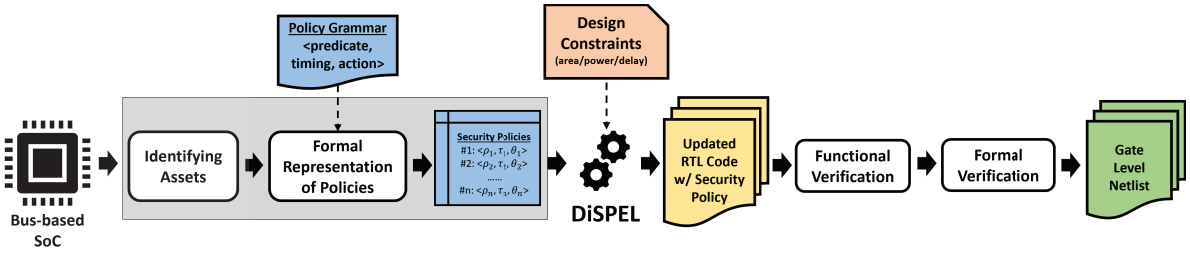



Fig. 1: **DiSPEL**: Proposed workflow for enforcing security policies represented in 3-tuple format for any bus-based SoC.

level language and lack flexibility and extensibility due to significant manual efforts involved in the workflow. Formalizing the security requirements and incorporating automation flow helps eliminate the dependence on security experts, reduces manual efforts, enhances response time, and enables a more proactive and effective approach to threat detection and mitigation efforts. Additionally, automation enables scalable and streamlined verification of the security policies with improved verification efficiency than manual testing efforts at different stages of the SoC development cycle in the zero-trust environment. Deb Nath et al. [5] proposed an architecture to facilitate the formal verification of security policies enforced in an SoC but has limitations in terms of generalization and applicability to a wider range of security scenarios for any bus-based SoC implementation.

In this paper, we propose **DiSPEL**, an automated framework for SoC security policy synthesis and distributed enforcement, which can address both bus-level security requirements involving multiple IPs and IP-level threats, arising from both software and hardware adversaries. For these threats, it incorporates low-overhead policy-based preventive measures and achieves significant speedup compared to manual efforts. Fig. 1 presents the overall workflow for the proposed framework. The major contributions of this work include:

- An automated framework, referred to as **DiSPEL**, for enforcing system-level security requirements via synthesizable security policies with minimal overhead.
- Specifying diverse security policies represented in a high-level 3-tuple representation with reconfigurability makes it extendable to any bus-based SoC for various threat models and trust assumptions.
- A hybrid approach to incorporate diverse bus-level security policies involving multiple IPs through a centralized policy module and IP-level policies through bus-level wrappers.
- An approach to ranking security policies using scores based on potential exploitation severity and employing a feedback path to discard lower-ranked policies to meet overhead constraints.

The remainder of the paper is organized as follows: Section II provides the background and an overview of the related work. Section III presents the methodology of the overall framework. The experimental results are described in Section IV. Finally, we conclude this paper in Section V.

## II. Background and Related Work

### A. Security Policy: Classification and Examples

Security Policies refer to guidelines, rules, or specifications designed to ensure the protection and integrity of assets within the SoC design. Security policies are put in place to prevent unauthorized access, detect and mitigate potential security threats, and maintain the overall security posture of the system. Security policies in modern SoC designs are often complex and developed in an ad-hoc manner due to their inherent intricacies and undergo refinement and modification throughout the system development process based on security requirements and product needs. Here are two representative examples of security policies for a typical System-on-Chip (SoC). It is important to note that these policies are purely for demonstrative purposes and do not represent the comprehensive set of security policies.

- *Example 1: Key authentication*
  The crypto engine will only provide actual keys in response to key access requests from other IPs if the requesting IP has been authenticated in test mode.
- *Example 2: Boot confidentiality*
  At boot time, all internal registers of the crypto engine are inaccessible to any IP.

Authors in [6], [7] presented a taxonomy of security policy classes in practical use, including Access Control, Information Flow, Liveness, and Time-of-Check vs. Time-of-Use (TOCTOU). In this work, we followed the same taxonomy for enforcing different security policies in any bus-based SoC. Fig. 2 illustrates a comprehensive SoC model with key components interconnected via a common bus: master IP, memory IP,

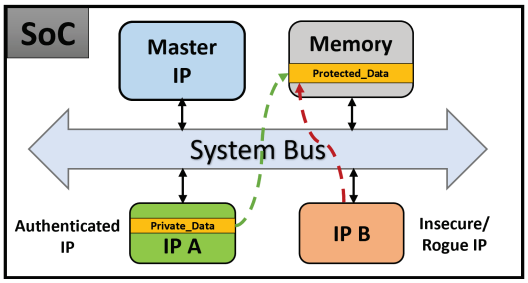



Fig. 2: Representation of a security threat: insecure/rogue IP is attempting to access private data during transit in a generalized bus-based SoC.

authenticated IP containing sensitive data, and an insecure 3PIP attempting unauthorized access. This illustration highlights a scenario analogous to *Example 1* and validates the need for implementing a robust security mechanism to safeguard sensitive data and uphold the confidentiality, integrity, and availability of the overall design.

### B. Trust Model

In this paper, the trust model assumes that an SoC integration house is trusted, acquiring 3PIPs from multiple vendors with varying levels of trustworthiness. While the IPs may not be inherently trustworthy, the bus wrappers and communication between the IP-level bus wrapper and the Security Policy Module are considered trusted. In modern SoCs, IPs are categorized by trust levels, and security policies are enforced to prevent untrusted entities from compromising secure assets or other system components. For instance, untrusted IPs like UART or JTAG, which interface with external entities and are susceptible to attacks, must have restricted access to sensitive data such as private cryptographic keys and user data achieved by enforcing appropriate access control policies.

### C. Threat Model

Effective asset protection in SoC designs requires the incorporation of security policies that outline protection requirements while considering potential adversaries' capabilities. Designers/Security Architects must recognize that adversaries can vary depending on the asset being protected. Defining and categorizing potential adversaries involves factors like physical access, observation, control, modification, and reverse-engineering capabilities, which enables the formulation of more targeted and robust security measures. In this work, we considered the threat models in the context of securing assets in an SoC, such as access control violations, information leakage, TOCTOU violations, etc.

### D. Previous Work

The early research on security policies in a computing system was primarily focused on information security [8] and developing a framework for analyzing the information flow and defining access control [9], [10]. However, the process of identifying and defining the policies was not found to be scalable for complex designs. X. Li et al. [11] proposed a typed language called SAPPER, which provides a synthesis framework for certain security policies. Some designers opted for other powerful languages, such as Property Specification Language or PSL [12], for defining and implementing security policies. Several researchers proposed another direction by representing security policies mapped to a specific class of Security Property. These properties can be tested statically using formal tools such as model checking [13], equivalence checking [14], and information flow tracking techniques [15]–[18], for detecting information leakage, especially in cryptographic designs that cause confidentiality and integrity property violation. A technique named Gate-Level Information Flow Tracking (GLIFT) has been introduced to detect and measure illegal flows of a corrupted value at the Boolean level [19]. Checking information flow helps to detect information leakage, which may infer the existence of Hardware Trojans [20]. Authors in [21] proposed an automated property generation for information flow properties for hardware designs. Farzana et al. [22] created a database of security properties and generated equivalent assertions to verify them using appropriate tool sets [23] for identifying vulnerabilities in a design. Authors in [24] proposed an automated mapping of security properties to cover new vulnerabilities at different abstraction levels. Authors in [25] proposed a formal approach for ensuring the integrity of security-critical operations for upholding access control demonstrated on OpenTitan's Earl Grey SoC. However, formal security properties are focused on identifying system vulnerabilities without enforcing the required code fixes, and they may not consistently scale well to meet the security demands of intricate SoC designs within acceptable overhead constraints.

Ray et al. [6] discussed about incorporating security policies becoming an area of significant research activities with the advancements of modern SoC designs involving third-party IPs. The research directed towards incorporating a dedicated security IP for providing system-level protection for SoCs with untrusted IPs emerged as one of the viable solutions against diverse attacks. Wang et al. [1] proposed an infrastructure IP (IIPS) for SoC security architecture but limited to preventing low-level hardware security vulnerabilities. Basak et al. [2], [26] extended the IIPS framework and presented a microcontroller-based framework for implementing certain classes of security policies. Nath et al. [3] proposed a Reconfigurable Security Policy Engine (RSPE), which can act as a smart security wrapper interfaced with the on-chip Design-for-Debug (DfD) interface for monitoring security-critical events and incorporating necessary mitigatory actions. Huang et al. [4] proposed an architecture called MSIPS that involves dedicated security IPs for enforcing security policies for protecting against H/W Trojan attacks, IP theft attacks, and some behavioral threats.

Existing mechanisms typically address threats occurring individually during runtime at either the architecture level or IP core level and incorporate dedicated security IPs to withstand a common set of security threats. However, the current methodologies lack the flexibility to represent different security requirements and define actionable specifications using a high-level formal representation for any bus-based design. Our proposed architecture **`DiSPEL`** is capable of generating a synthesizable centralized module with minimal overhead for enforcing security requirements through policies from given user specifications. Table I provides a comparative analysis of **`DiSPEL`** with existing solutions.

## III. Methodology

The proposed framework combines both manual and automated steps, starting with asset identification by analyzing each IP in the SoC and the underlying threat model. Once identified, designers/architects can create security policies for relevant IPs to secure the assets. Security policies are represented in a 3-tuple format for readability and interpretation:

TABLE I: Comparison with existing solutions & Scope of the current work.

| Proposed Solutions | Working with Bus-Level SoC? | Centralized Policy Engine? | IP-Level Policies? | Support for Syn/Non-Syn Constructs? | Policies in Formal Representation? | Generalized Policies for Diverse Needs? | Automated Flow? |
|---|---|---|---|---|---|---|---|
| IIPS [1] | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| E-IIPS [2] | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| RSPE [3] | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| MSIPS [4] | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ |
| **DiSPEL*** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

* current work

$< predicate, timing, action >$. Each security policy includes a *predicate* indicating specific conditions based on observable internal signals as a Boolean function, *timing* information related to the global clock, and an *action* specifying the task of asserting or de-asserting bus signals. **DiSPEL** automatically enforces security policies without altering the existing SoC design, thus alleviating the complexity. The workflow of the proposed framework is shown in Fig. 3.

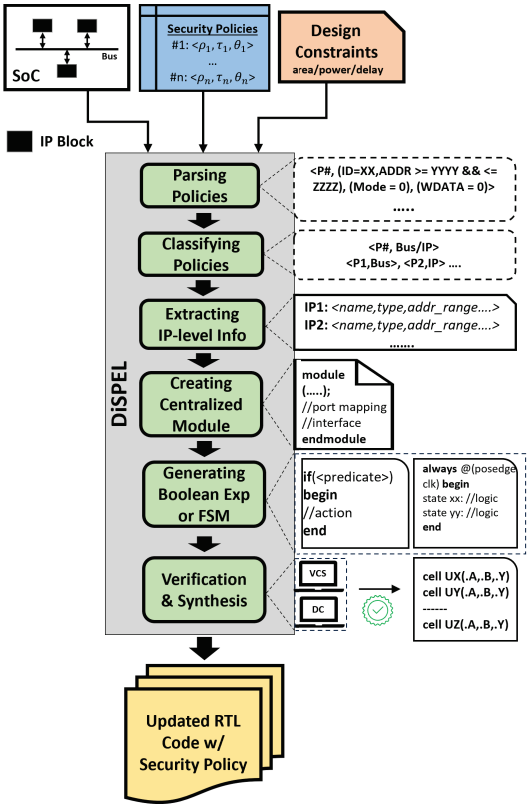



Fig. 3: Major steps in the **DiSPEL** workflow for enforcement of security policies for any bus-based SoC.

The automated tool, DiSPEL, operates through six primary steps, outlined as follows:

1) *Parsing Security Policies*: **DiSPEL** employs a regular expression-based parsing of policies from a .json file to identify the predicate, timing, and action fields for each policy. Any representation inconsistencies trigger error generation. The parsed information is stored in the appropriate data structure individually for each policy, with default values assigned for undefined fields like timing/mode (if any).
2) *Classifying Policies*: **DiSPEL** categorizes the policies as bus-level or IP-level and distinguishes them as synthesizable or non-synthesizable. For synthesizable policies, it generates the required logic for policy enforcement. Non-synthesizable policies result in the generation of SystemVerilog Assertions for verification at the specified IP or bus level. **DiSPEL** supports intricate clock cycle requirements involving FSMs for sequential events. To enhance user convenience when defining policies in tuple format, we have included support for several common keywords (refer to Table II) that are translated to equivalent bus signal names by the tool.
3) *Extracting IP-level Information*: **DiSPEL** relies on information about IP level configuration curated by the designer/architect/developer to precisely analyze, process, and generate appropriate code for enforcing security policies. The IP level configuration for each master and slave IP includes the IP name, IP type, address range, base/starting/ending addresses, and starting & ending data markers. The configuration file should also incorporate supplementary details, such as the global clock name, reset name, bus protocol, etc.
4) *Creating a Centralized Security Module*:
After extracting IP-level configurations, **DiSPEL** generates a centralized module in SystemVerilog to control signals between these IPs and the bus interconnect (refer to Fig. 4). The centralized module ensures that security requirements are met by enforcing relevant policies by monitoring data flow between IP interfaces and the bus, protecting sensitive assets.

TABLE II: List of supported keywords by **DiSPEL** tool.

| Keywords | Eqv. Bus Signals* | |
|---|---|---|
| | AXI4 | Wishbone |
| read_address/read address/raddress | S_AXI_ARADDR | wb_adr_i |
| write_address/write address/waddress | S_AXI_AWADDR | wb_adr_o |
| read_data/read data/rdata | S_AXI_RDATA | wb_dat_i |
| write_data/write data/wdata | S_AXI_WDATA | wb_dat_o |
| strobe/strb/wstrb | S_AXI_WSTRB | wb_sel_o |
| write_ready/write ready/wready | S_AXI_WREADY | wb_ack_i |
| read_ready/read ready/rready | S_AXI_RREADY | - |
| address_ready/address ready/arready | S_AXI_AWREADY | - |
| address_valid/address valid/arvalid | S_AXI_AWVALID | wb_we_o |
| write_valid/write valid/wvalid | S_AXI_WVALID | wb_stb_o |
| read_valid/read valid/rvalid | S_AXI_RVALID | - |

*Note: The representation of Eqv. Bus Signals are based on the MIT-CEP SoC benchmark and might need to be altered for other benchmarks accordingly.

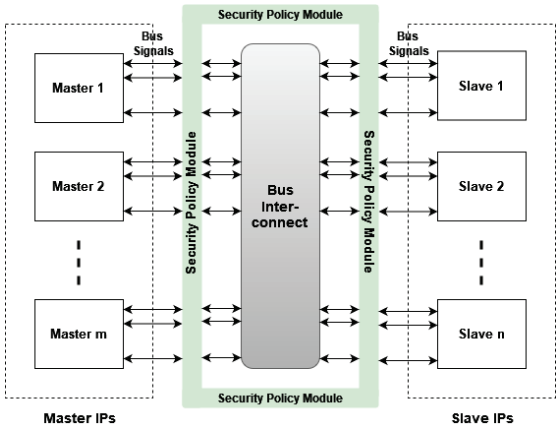



Fig. 4: Placement of centralized policy module between IPs and bus interconnect in any bus-based SoC.

5) *Representation using Boolean Expression or Finite State Machine (FSM)*: **DiSPEL** evaluates security policies by identifying their conditions and actions, converting them into SystemVerilog conditional statements. For complex timing and sequential behavior, it uses FSM-based transitions and flags. Leveraging master/slave IP configurations, it associates IP modules with policies and assigns signal values to enforce security, protecting SoC assets.
6) *Verification & Synthesis*: The generated SystemVerilog code by **DiSPEL** is verified using commercial Electronic Design Automation (EDA) tools to ensure correct functionality using simulation and formal verification. If successful, the code is synthesized into a gate-level netlist for subsequent stages in the design flow.

The **DiSPEL** framework uses a feedback-driven approach to meet design constraints (area, power, delay). All relevant security policies are ranked based on their potential exploitation severity. The initial synthesis step includes all security policies. If constraints are not met, lower-ranked policies are removed, and the SystemVerilog code is resynthesized. For instance, a policy protecting a 32-bit secure asset in a crypto IP is given a higher score than one safeguarding a 1-bit value in an arithmetic unit. The score is calculated for each policy as follows:

$$Score = f(IP_{type}, \#Bits_{secured}, Attack_{type}, Policy_{type}) \quad (1)$$

**DiSPEL** iteratively removes 10% of policies based on scores until constraints are met. Table III shows policy scores reflecting severity, with each policy identified by a unique $Policy_{id}$ ($\phi_i$) corresponding to a specific $IP_{type}$ in the SoC. $\#Bits_{secured}$ signifies the data protected by each policy. $Attack_{type}$ includes C, I, and A violations, while $Policy_{type}$ indicates bus or IP-level enforcement.

Algorithm 1 describes the whole workflow of the DiSPEL automated tool in which we demonstrate the process of enforcing policies through a centralized policy module or appending to the corresponding bus-level wrapper for IP-level policies. The inputs to the DiSPEL framework are described as follows:

TABLE III: Normalized scores for different security policies.

| $Policy_{id}$ | $IP_{type}$ | $\#Bits_{secured}$ | $Attack_{type}$ | $Policy_{type}$ | $Score$ |
|---|---|---|---|---|---|
| $\phi_1$ | Crypto | 128 | C,I,A | Bus | 19.2 |
| $\phi_2$ | Hashing | 64 | C,I,A | Bus | 7.68 |
| $\phi_3$ | Memory | 32 | C,A | IP | 0.96 |
| $\phi_4$ | DSP | 16 | A | IP | 0.16 |

- $\mathbb{F}_{soc}$: Denotes SoC configuration file, presented in JSON format, containing implementation details for each IP and the underlying bus protocol.
- $\Phi$ : Denotes the list of Security Policies such that, $\forall i, \phi_i = < predicate(\rho), timing(\tau), action(\theta) >$

The DiSPEL tool generates the centralized policy module ($\mathcal{F}$) and updated bus-level wrapper ($\mathcal{W}'$) with security policies incorporated in them. Here are descriptions of some of the methods used in the algorithm:

- ***create_security_policy_module()***: Creates the basic structure of the centralized module, including interfaces, port definitions, etc.
- ***classify_policy()***: Returns *true* if the predicate contains IP-level information inferring to an IP-level policy.
- ***is_syn()***: Returns *true* if the policy is synthesizable. A Synthesizable policy involves relevant bus signals, timing requirements (optional), and specific actions for enforcement.
- ***is_sequential()***: Returns *true* if the predicate consists of a sequence of events involving bus-level or IP-level signals necessitating the incorporation of FSM.
- ***is_clock_cycles()***: Returns *true* if the policy includes a specific number of clock cycles to be counted in the predicate or timing tuple.
- ***create_assertion()***: Generates assertions for non-synthesizable policies, which include specific conditions in predicate and timing tuples for verification purposes.
- ***replace_keywords()***: Replaces the keywords with the respective bus signal names as described in Table II.
- ***extract_id_flag()***: Returns the identifier (preferably an integer) of the respective master or slave corresponding to the policy being parsed.
- ***create_fsm()***: Generates the required FSM block for accommodating the necessary logic. It returns the name of the register that signifies the occurrence of the event when the condition defined in the predicate is satisfied.
- ***create_cond()***: Generates the required conditional block using *if-else* statements and combining all the condition(s) specified in the predicate and timing tuple.

## IV. Experimental Results

### A. SoC Benchmark used for Testing & Evaluation

This section provides an overview of the experimental setup and analysis of the obtained results to evaluate the performance of the proposed framework. To assess its effectiveness, we selected the CEP (Common Evaluation Platform)[1], an open-source SoC benchmark developed by MIT, as our testing and evaluation platform. The CEP benchmark comprises various

[1]https://github.com/mit-ll/CEP.git

**Algorithm 1:** Enforcement of Security Policies

**Input:** $\mathbb{F}_{soc}$, $\Phi$
**Output:** $\mathcal{F}$, $\mathcal{W}'$

$module\_def, master\_block, slave\_block =$ **create_security_policy_module(**$\mathbb{F}_{soc}$**)**
**for** *each* $\phi_i \in \Phi$ **do**
  $syn\_flag = false, fsm\_flag = false$
  $\{\rho, \tau, \theta\} <= split(\phi_i)$
  $ip\_level =$ **classify_policy**$(\phi_i)$
  **if** ***is_syn***$(\rho)$ **then**
    $syn\_flag = true$
    **if** ***is_sequential***$(\rho)$ *or* ***is_clock_cycles***$(\tau)$ **then**
      $fsm\_flag = true$
    **end**
  **end**
  **else**
    $assertion$ = **create_assertion**$(\rho, \tau, \mathbb{F}_{soc})$
    $\mathcal{W}'_{id} = \mathcal{W}_{id}.append(id, assertion)$
  **end**
  **if** $syn\_flag$ **then**
    **if** $\rho$***find***$(keywords)$ **then**
      $\rho' <=$ **replace_keywords**$(\rho)$
      $id, m\_flag, s\_flag =$ **extract_id_flag**$(\rho')$
      $\sigma = split(\rho')$
    **end**
    **else**
      $id, m\_flag, s\_flag =$ **extract_id_flag**$(\rho)$
      $\sigma = split(\rho)$
    **end**
    **if** $fsm\_flag$ **then**
      flag_name = **create_fsm(**$\sigma, \tau, \theta, \mathbb{F}_{soc}$**)**
      **if** $ip\_level$ **then**
        $\mathcal{W}'_{id} = \mathcal{W}_{id}.append(id, flag\_name)$
      **end**
      **else**
        $(m\_flag)$?
        master_block.$append(id, flag\_name)$ :
        slave_block.$append(id, flag\_name)$
      **end**
    **end**
    **else**
      block = **create_cond(**$id, \sigma, \tau, \theta$**)**
      **if** $ip\_level$ **then**
        $\mathcal{W}'_{id} = \mathcal{W}_{id}.append(block)$
      **end**
      **else**
        $(m\_flag)$? master_block.$append(block)$
        : slave_block.$append(block)$
      **end**
    **end**
  **end**
**end**
$\mathcal{F}.write(\{module\_def, master\_block, slave\_block\})$
*return* $\mathcal{F}, \mathcal{W}'$

components, including the Mor1kx OpenRISC processor, which acts as the Master IP, and a set of Slave IPs with diverse functionalities. These Slave IPs consist of several cryptographic modules, such as AES, DES3, RSA, MD5, and SHA-256, and digital signal-processing modules like DFT, IDFT, FIR, IIR, and a GPS module. Additionally, there are certain modules like JTAG, UART, etc., which facilitate external communication with the user. It should be noted that these modules may serve as potential attack surfaces and are, therefore, considered untrusted IPs. Fig. 5 illustrates a basic block diagram of the SoC benchmark used in our experiment. We have categorized the Crypto and DSP modules along with the open-source RISC processor as the master IP and the potentially insecure IPs communicating through the AXI4 bus interconnect.

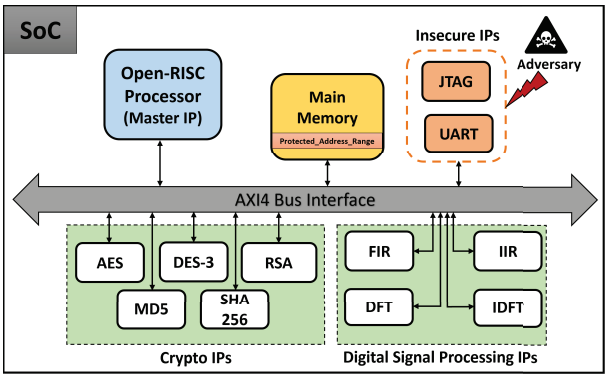



Fig. 5: A representative block diagram of the test SoC benchmark with master & slave IPs along with insecure IPs connected through a common bus interface.

### *B. An Example Security Policy: Representation & Enforcement*

To provide a comprehensive insight into the automated generation process of synthesizable SystemVerilog code with security policy we will illustrate one such example policy below.
Policy #: The processor (Master) is NOT ALLOWED to write data in between a specific ADDRESS RANGE of the RAM (Slave) in the user mode.

**Representation in 3-tuple format:**

```
policy.json
{
  "Policy#":
  {
    "predicate": "address >= 0x0001dfa4 and address
    <= 0x0001ffac",
    "timing": "mode = 0",
    "action": "write_data = 0"
  }
}
```

The enforcement of the security policy through the DiSPEL tool flow is as follows:

- The tool categorizes each Policy# as synthesizable or non-synthesizable based on its representation using ***is_syn()*** method. The following steps will be carried out since the policy is marked as synthesizable. The value of *syn_flag* will be set to *true*.

TABLE IV: Example security policies with their respective policy representation and SystemVerilog constructs.

| Security Policy | IPs Involved | Policy Representation in 3-tuple format | SystemVerilog Constructs |
|---|---|---|---|
| Policy **P1**: The Master must restrict the write data operation for some specific addresses/address range of the Slave. | OpenRISC Processor (Master) Memory/AES/DES3/SHA-256 etc. (Slave) | predicate : (write_address >= 9300000c and write_address <= 93000010)<br>timing : "mode = 0"<br>action : "write_data = 0" | if(master[i].aw_addr >= 32'h9300000c && master[i].aw_addr <= 32'h93000010)<br>begin<br>master_spw[i].w_data = 0 ;<br>end |
| Policy **P2**: The Master is not allowed to read data from a specific address range of the Slave. | OpenRISC Processor (Master) Memory/AES/DES3/SHA-256 etc. (Slave) | predicate : (slave_no = 1), (read_address >= 93000004 and read_address <= 93000008)<br>timing : "mode = 0"<br>action : "rdata = 0" | if ((i==1) && (slave_spw[i].aw_addr >= 0x93000004 && slave_spw[i].aw_addr <= 0x93000008))<br>begin<br>slave[i].r_data = 0;<br>end |
| Policy **P3**: The untrusted Slave(s) are not allowed to access the bus when Secret Keys are being sent to Slave Crypto Module. | AES/DES3/RSA etc. (Trusted Slave) JTAG/UART etc. (Untrusted Slave) | predicate : (slave_no = 4 or 5), (write_address >= 93000014 and write_address <= 93000028)<br>timing : "mode = 0"<br>action : "wdata = 0" | if ((!(i == 4 \|\| i == 5)) && (slave_spw[i].aw_addr >= 0x93000014 && slave_spw[i].aw_addr <= 0x93000028))<br>begin<br>slave[i].w_data = 0;<br>end |
| Policy **P4**: If any particular Slave runs for more than 'n' number of cycles then discard the result since the slave might be compromised. | AES/DES3/RSA/SHA-256/MD5 etc. (Slave) | predicate : (slave_no = 2), (clock_cycles >1000)<br>timing : "mode = 0"<br>action : "r_data = 0" | if ((i==2) && (flag==1)) //flag is a reg that //denotes if the condition is reached<br>begin<br>slave[i].r_data = 0;<br>end<br>//FSM to Count # cycles while in Operation<br>...FSM States...<br>if(count >1000) begin<br>flag <= 1;<br>curr_state <= 1'b0;<br>end |

TABLE V: Enforcement of a specific security requirement through security property and security policy.

| Abstraction Level | IPs Involved | Security Property | Security Policy |
|---|---|---|---|
| RTL | AES (Slave IP) | no_key_leakage_to_output := assert (CT != linear_func(K)) | if($countones(ct[127:96] ^ key[0])<8 \|\| $countones(ct[127:96] ^ key[1])<8 \|\| $countones(ct[127:96] ^ key[2])<8 \|\| $countones(ct[127:96] ^ key[3])<8 \|\| $countones(ct[127:96] ^ key[4])<8 \|\| $countones(ct[127:96] ^ key[5])<8))<br>wb_dat_o <= 32'b0;<br>else<br>wb_dat_o = ct[127:96];<br>//repeat for ct[95:64], ct[63:32] & ct[31:0] |

- The predicate will be translated as:

```
((<address_var> >= 32'h0001dfa4) && (<address_var
    > <= 32'h0001ffac))
```

- The timing tuple defines user mode and is converted to an equality comparison statement as:

```
 (<mode_reg_val> == 0)  //defines user mode
```

- The tool will generate the conditional statement in SystemVerilog to enforce the security policy through ***create_cond()*** method as:

```
if ((master.aw_addr >= 32'h0001dfa4) && (master.
    aw_addr <= 32'h0001ffac)) && (reg_mode == 1'
    b0))
begin
    //action
end
```

- Based on the action definition, the respective signals will be updated. In this scenario, the signal for writing data will be adjusted as follows:

```
    master_out.wdata <= 32'h0;  //action
```

Table IV represents a few example security policies applicable for either master or slave or both IPs and the policy representation in 3-tuple format with corresponding SystemVerilog construct for enforcement of each policy. Fig. 6 demonstrates the enforcement of the security policy from a security requirement that writing data is not allowed in the protected address region by the Master. The diagram shows a snapshot of the communication between the Master IP (Processor) as the sender and the Slave IP (Memory) as the receiver, with other slave IPs connected through the bus interconnect. For each IP, a number of bus signals and the respective values are mentioned at that time instance. The centralized module modifies the current bus transaction by updating the values of $w\_data$ to 0x0000000 and $w\_valid$ to 0, which will restrict the writing of data in memory and hence comply with the security requirements.

TABLE VI: Centralized Module Implementation Results after Synthesis.

| #Policies | Implementation Results | | |
|---|---|---|---|
| | **Area** ($\mu m^2$) | **Delay** ($ns$) | **Power** ($mW$) |
| 10 | 17236.53 | 1.88 | 4.7401 |
| 20 | 22835.65 | 1.88 | 3.4432 |
| 30 | 24556.05 | 1.89 | 4.7401 |

### C. Security Properties vs. Security Policies

In the context of security verification, a security property serves as a statement to examine inferences, prerequisites, assumptions, and expected behaviors within a design. This statement can take the form of an assertion or a cover statement. Assertions are utilized to verify correct functioning and detect invalid events that may induce potential threats, while cover statements help in obtaining coverage information during the validation process of the design. Security properties can be immediate or concurrent, with the latter being more potent for expressing complex events. Designers commonly use assertion languages like PSL and SystemVerilog Assertions, employing

TABLE VII: Overhead Analysis after Synthesis of Different IPs after Security Policy Enforcement.

| IP | #Policies | Original Circuit | | | Modified Circuit | | | PPA Overheads* (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Area ($\mu m^2$) | Delay ($ns$) | Power ($mW$) | Area ($\mu m^2$) | Delay ($ns$) | Power ($mW$) | Area | Delay | Power |
| AES | 10 | 1240769.98 | 4.47 | 108.49 | 1240802.51 | 4.74 | 107.66 | 0.02 ↑ | 6.04 ↑ | −0.77 ↓ |
| DES3 | 10 | 23797.82 | 4.67 | 5.27 | 26085.02 | 4.71 | 5.57 | 9.61 ↑ | 0.85 ↑ | 5.72 ↑ |
| SHA256 | 10 | 106776.15 | 6.67 | 13.05 | 104361.34 | 6.87 | 13.16 | −2.26 ↓ | 2.99 ↑ | 0.81 ↑ |
| MD5 | 5 | 116501.73 | 6.2 | 12.99 | 119433.29 | 6.48 | 12.28 | 2.51 ↑ | 4.51 ↑ | −5.54 ↓ |
| RSA | 5 | 2458595.43 | 8.7 | 269.89 | 2459273.58 | 8.71 | 269.92 | 0.02 ↑ | 0.11 ↑ | −0.01 ↓ |

*Note: The negative overhead values observed are caused by area optimization under unconstrained synthesis and vector-less power estimation.

logic representations at the temporal level like LTL and CTL [27], [28] to describe design behaviors. Various tools and methodologies, such as model-checking for integrity properties and information flow tracking for confidentiality, are used to verify security properties.

Unlike non-synthesizable security properties, the proposed **DiSPEL** tool generates synthesizable code in SystemVerilog for enforcing security policies in bus-based SoC architecture. The synthesizable code can be verified using simulation using any industry standard simulator, while it can be synthesized to a gate-level netlist using standard EDA tools. Hence, the proposed architecture can be easily integrated into both ASIC and FPGA flow methodologies of any bus-based SoC design. Table V demonstrates the following security requirement using security property and in terms of security policy based on the SystemVerilog conditional statement generated by **DiSPEL** tool.

*Security Requirement*: The secret key (K) should not be leaked (partially or fully) through Ciphertext (CT).

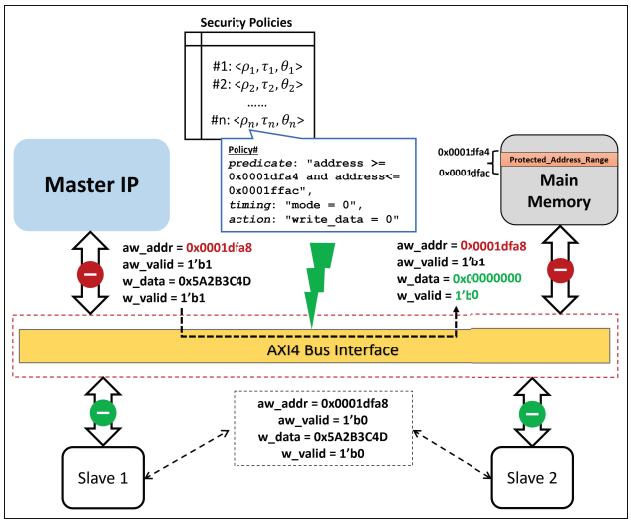



Fig. 6: Example of security policy enforcement by the centralized module involving multiple IPs in a bus-based SoC.

### D. Overhead Analysis

We obtained representative overhead values for IPs by synthesizing them using Synopsys Design Compiler for the 130nm SkyWater Open Source PDK[2]. Table VI shows the area, delay, and power consumption reported by Synopsys Design Compiler for the centralized policy module implementation with different numbers of policies enforced. Table VII provides a comparative analysis of PPA [Power, Performance (delay) and Area] values between the original and modified designs along with the overhead calculated in % for various IPs with varying numbers of IP-level policies. The results indicate that overheads are generally minimal under default synthesis settings, and in some cases, they decrease after implementing the security policy module due to internal optimizations. Power values reported by the synthesis tool may not reflect actual design functionality under default tool settings; a vector-based approach is required to obtain the correct power values. Our approach is performance-efficient, enabling policy enforcement, verification, and synthesis in just a few minutes. For example, 10 policies were enforced in an open-source *AES* benchmark (∼300K gates) followed by verification and synthesis in under 7 minutes. Therefore, **DiSPEL** imposes minimal overhead and is practical for implementation. Fig. 7 presents statistical data indicating the enforcement of security policies from various classes by **DiSPEL** in the overhead analysis.

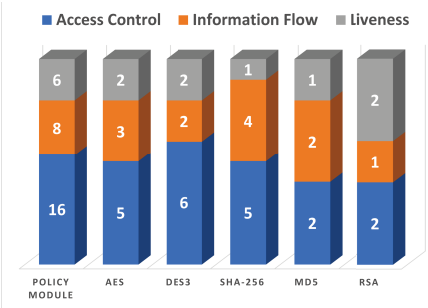



Fig. 7: Different modules with varying numbers of security policies belonging to different classes.

Below is an illustrative example to quantify the speedup achieved by **DiSPEL** compared to manual efforts.

Let us assume,

- $MH_{manual}$ represents the man-hours required for manual efforts before automation.
- $MH_{automation}$ represents the man-hours required for the same task after automation.
- P denotes the performance improvement in percentage.

$$P = \frac{(MH_{manual} - MH_{automation})}{MH_{manual}} * 100 \qquad (2)$$

[2]https://github.com/google/skywater-pdk

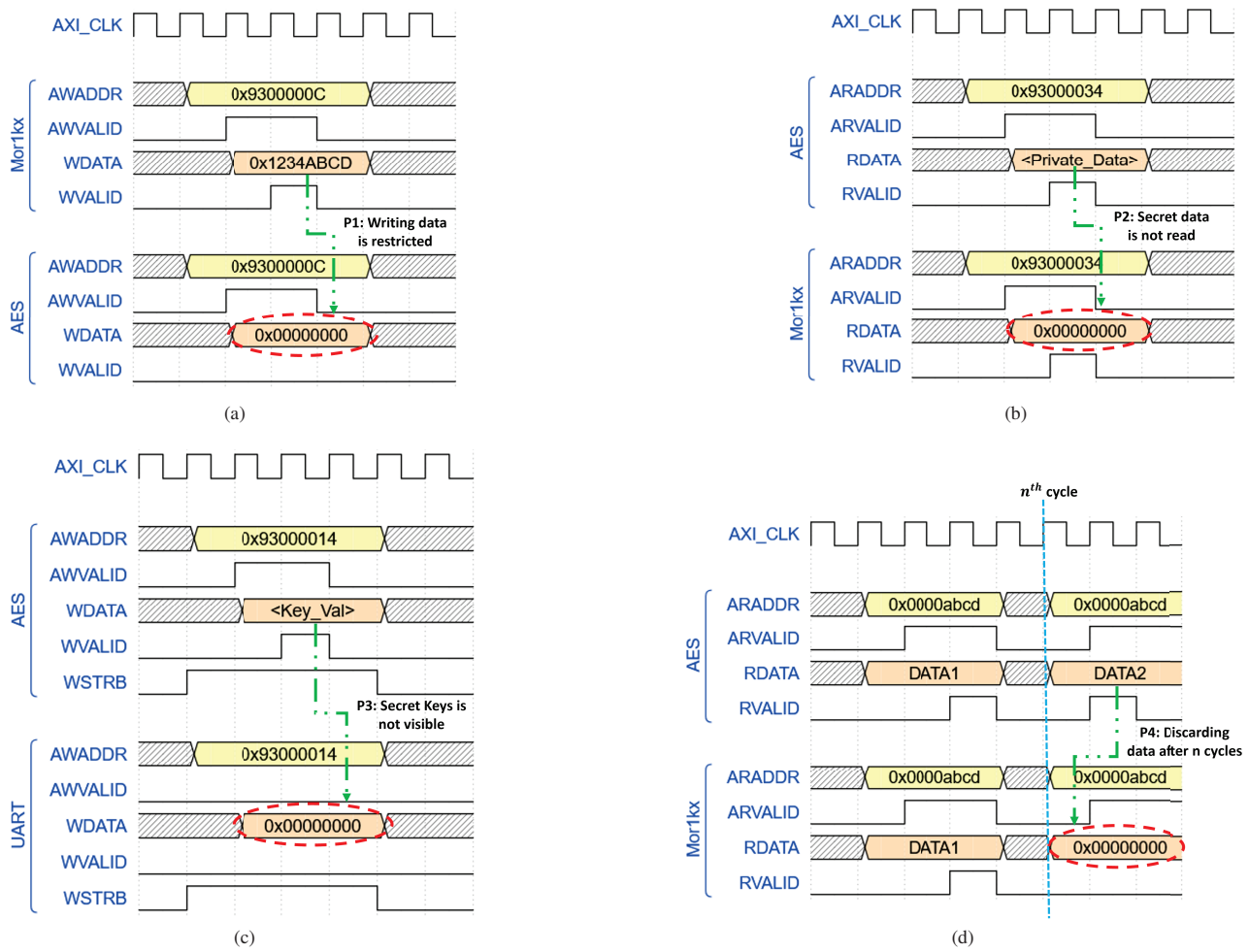



Fig. 8: Timing diagrams illustrating the enforcement of security policies: (a) P1: Mor1kx (Master) and AES IP (Slave), (b) P2: AES IP (Slave) and Mor1kx (Master), (c) P3: AES IP (Slave) and UART (Slave), (d) P4: AES IP (Slave) and Mor1kx (Master).

$$Speedup = \frac{MH_{manual}}{MH_{automation}} \tag{3}$$

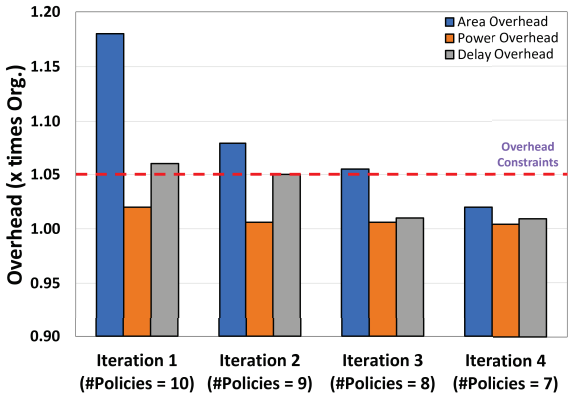
In a real-world scenario, manually identifying and fixing vulnerabilities would require a few weeks by a number of security experts for a complex bus-based SoC design. Assuming an approximate value for $MH_{manual}$ as 800, accounting for 5 personnel working 8 hours a day over 20 days, automation emerges as a transformative solution. **DiSPEL** streamlines the process by generating security policies based on user specifications and automatically enforcing the necessary code fix as required. Let us assume an approximate value for $MH_{automation}$ to be 40, considering 1 personnel working 8 hours/day for 5 days. Consequently, the achieved speedup through automation is 20, accompanied by a notable performance improvement of 95%.

### E. Waveform Analysis

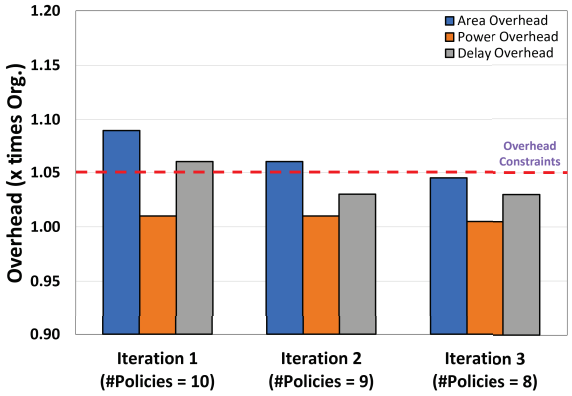
We generated waveforms using Synopsys VCS (Verilog Compiler Simulator) and viewed them through the DVE (Discovery Virtualization Environment) interface to validate the effectiveness of enforcing a security policy aimed at protecting secure assets during transit. Fig. 8 displays timing diagrams illustrating the successful enforcement of the security policies outlined in Table IV.

### F. Meeting Design Overhead Constraints

**DiSPEL** ensures overhead constraints in terms of area, power, and delay are met by prioritizing policies based on severity and using a feedback approach. If constraints are not satisfied, 10% of the lowest-ranked policies are eliminated and re-synthesized iteratively until the constraints are met. Fig. 9 shows multiple iterations for the SHA-256 and DES3 modules from our test SoC benchmark. The overhead constraints were set to less than 5% (<1.05 times original) for area, power, and delay. The bar graph illustrates how the first few iterations violated the overhead constraints, but all constraints were met after policy adjustments and re-synthesis iteratively. The effectiveness of **DiSPEL** tool was further tested




(a)




(b)

Fig. 9: Overhead values represented as x times the original Design and multiple iterations to meet the overhead constraints for (a) SHA-256 Module (b) DES3 Module.

by applying time constraints-based synthesis and experimenting with different IP modules in multiple iterations until the allowed constraints were met. It's crucial to emphasize that we are evaluating a trade-off between achieving the desired security level and adhering to overhead constraints by iteratively discarding lower-ranked policies. The underlying assumption is that the severity of exploitation through these lower-ranked policies is comparatively lower, resulting in a reduced impact on overall security. However, there might be a situation where the overhead constraints are violated, yet all the highest-scoring threats must be addressed. Under such circumstances, it becomes viable to consider sacrificing the least critical overhead constraint to enhance security further. The trade-off between policy enforcement and design constraints primarily follows a balanced strategy and can be adjusted depending on the circumstances to achieve the highest security.

### G. Simulation & Formal Verification Results

We demonstrate security policy effectiveness through simulation and ensure correctness with formal verification. Synopsys VCS is used for simulation using bare metal code and unit-level testing for individual IPs, and Formal Property Verification (FPV) is performed using Cadence JasperGold. Thorough validation was performed to ensure compliance with specified security requirements. Table VIII shows verification results of the example policies from Table IV for different IPs in our test SoC benchmark. For each policy, the corresponding predicate and timing conditions are converted into a coverage statement. For the example security policy in Section IV-B, the coverage statement (shown below) generated by DiSPEL was verified in less than 0.1 seconds with *engine_mode* set to 'auto' and reported as 'Covered' by JasperGold, confirming the correctness of policy enforcement.

```
% cover -name {<embedded>::test_cov} {wb_adr_i >= 32'
    h0001dfa4 && wb_adr_i <= 32'h0001ffac && mode_reg
     == 1'b0} -update_db;
% prove -bg -task {<embedded>}
```

TABLE VIII: Verification after Security Policy Enforcement.

| IPs Involved | Policy | Verification Results | |
|---|---|---|---|
| | | Simulation | Formal Verification |
| $\mu$P, AES/DES3/SHA-256 | P1,P2 | Passed | Covered |
| $\mu$P, Memory | P1,P2 | Passed | Covered |
| AES/DES3, UART | P3 | Passed | Covered |
| SHA-256, UART | P3 | Passed | Covered |
| AES/DES3/SHA-256 | P4 | Passed | Covered |

## V. Conclusion & Future Work

Security policies play a critical role in modern SoCs to safeguard on-chip assets. In this paper, we have presented **DiSPEL**, an automated framework for implementing diverse security policies in SoCs in a systematic and distributed at modest hardware overhead. We have adopted a formal representation of the policies using a simple grammar. The **DiSPEL** workflow facilitates security requirement definition and simplifies security policy enforcement. **DiSPEL** employs a hybrid approach for implementing the policies that include a centralized security module for bus-level policies and IP-level wrappers for IP-specific security requirements. **DiSPEL** adopts a feedback-based approach for meeting design overhead constraints by ranking the policies based on the severity of exploitation and discarding lower-ranked policies iteratively until the constraints are met. The proposed framework is demonstrated to be scalable and robust, capable of supporting complex designs involving various IPs in bus-based SoCs. We have validated the functional correctness of the framework through simulation and formal verification on a set of open-source SoCs. The experimental results demonstrate that **DiSPEL** incurs minimal area, power, and delay overheads for practical SoCs. Future work will expand support for different SoC configurations and bus protocols and can be further extended for NoC configurations supporting diverse classes of security policies. Another interesting direction would be exploring the optimal trade-off between accommodating design overhead constraints and the selective removal of policies to achieve maximum security.

## REFERENCES


[1] X. Wang, Y. Zheng, A. Basak, and S. Bhunia, "IIPS: Infrastructure IP for Secure SoC Design," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2226–2238, 2015.

[2] A. Basak, S. Bhunia, and S. Ray, "A flexible architecture for systematic implementation of SoC security policies," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 536–543.

[3] A. P. D. Nath, S. Ray, A. Basak, and S. Bhunia, "System-on-Chip Security Architecture and CAD Framework for Hardware Patch," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, ser. ASPDAC '18. IEEE Press, 2018, p. 733–738.

[4] Z. Huang and Q. Wang, "Enhancing Architecture-level Security of SoC Designs via the Distributed Security IPs Deployment Methodology," *J. Inf. Sci. Eng.*, vol. 36, pp. 387–421, 2020.

[5] A. P. Deb Nath, S. Bhunia, and S. Ray, "ArtiFact: Architecture and CAD Flow for Efficient Formal Verification of SoC Security Policies," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 411–416.

[6] S. Ray and Y. Jin, "Security policy enforcement in modern SoC designs," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 345–350.

[7] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2018.

[8] S. J. Greenwald, "Discussion Topic: What is the Old Security Paradigm?" in *Proceedings of the 1998 Workshop on New Security Paradigms*, ser. NSPW '98, New York, NY, USA, 1998, p. 107–118.

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

[10] J. T. Haigh and W. D. Young, "Extending the Noninterference Version of MLS for SAT," *IEEE Trans. Softw. Eng.*, vol. 13, no. 2, p. 141–150, feb 1987.

[11] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A Language for Hardware-Level Security Policy Enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, NY, USA, 2014, p. 97–112.

[12] M. Grüninger and C. Menzel, "The Process Specification Language (PSL) Theory and Applications," *AI Mag.*, vol. 24, no. 3, p. 63–74, sep 2003.

[13] J. Rajendran, V. Vedula, and R. Karri, "Detecting malicious modifications of data in third-party intellectual property cores," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.

[14] F. Farahmandi and P. Mishra, "Automated Debugging of Arithmetic Circuits Using Incremental Gröbner Basis Reduction," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017, pp. 193–200.

[15] W. Hu, B. Mao, J. Oberg, and R. Kastner, "Detecting Hardware Trojans with Gate-Level Information-Flow Tracking," *Computer*, vol. 49, no. 8, pp. 44–52, 2016.

[16] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property Specific Information Flow Analysis for Hardware Security Verification," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[17] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 254–259.

[18] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the Complexity of Generating Gate Level Information Flow Tracking Logic," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 1067–1080, 2012.

[19] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 109–120. [Online]. Available: https://doi.org/10.1145/1508244.1508258

[20] J. Rajendran, A. M. Dhandayuthapany, V. Vedula, and R. Karri, "Formal Security Verification of Third Party Intellectual Property Cores for Information Leakage," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 547–552.

[21] C. Deutschbein, A. Meza, F. Restuccia, M. Gregoire, R. Kastner, and C. Sturton, "Toward Hardware Security Property Generation at Scale," *IEEE Security & Privacy*, vol. 20, no. 3, pp. 43–51, 2022.

[22] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "SoC Security Verification using Property Checking," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[23] S. Aftabjahani, R. Kastner, M. Tehranipoor, F. Farahmandi, J. Oberg, A. Nordstrom, N. Fern, and A. Althoff, "Special Session: CAD for Hardware Security - Automation is Key to Adoption of Solutions," in *2021 IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–10.

[24] B. Ahmed, F. Rahman, N. Hooten, F. Farahmandi, and M. Tehranipoor, "Automap: Automated mapping of security properties between different levels of abstraction in design flow," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[25] D. Mehmedagić, M. R. Fadiheh, J. Müller, A. L. D. Antón, D. Stoffel, and W. Kunz, "Design of access control mechanisms in Systems-on-Chip with formal integrity guarantees," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2779–2796.

[26] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security Assurance for System-on-Chip Designs With Untrusted IPs," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.

[27] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

[28] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, p. 244–263, apr 1986.