The Road Not Taken: eFPGA Accelerators Utilized for SoC Security Auditing

Mridha Md Mashahedur Rahman, Shams Tarek, Kimia Zamiri Azar, Mark Tehranipoor, and Farimah Farahmandi
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA.
E-mail: {mrahman1, shams.tarek, k.zamiriazar}@ufl.edu, {tehranipoor, farimah}@ece.ufl.edu

***Abstract*—To meet the demands of diverse and rapidly evolving markets, system-on-chips (SoCs) are becoming more complex in size and functionality. More IPs and hardware accelerators are required to support a varied set of applications with faster response. In recent years, there has been a growing trend of using reconfigurable and adaptable hardware for compute-intensive kernels, e.g., neural networks, crypto-engines, and blockchains. Hence, embedded FPGA (eFPGA) technology has emerged as a standard solution incorporated into the SoC to enhance computational performance and provide reconfigurability. However, with the increasing complexity and size of modern SoCs, coupled with the integration of third-party IPs (3PIPs) and accelerators, ensuring the information security, i.e., *integrity*, *confidentiality*, and *availability*, of critical and sensitive data has become more challenging than ever before. Thus, a sustainable and upgradable security auditing infrastructure has become a necessity. This paper extends *EnSAFe*, a framework specially crafted to streamline security policy auditing while enabling upgradability within designs that leverage eFPGA-based accelerators. The *EnSAFe* framework enables signal monitoring in a plug-and-play fashion, and the monitoring core logic is mapped onto the eFPGA accelerator component with minimal overhead. We extend *EnSAFe* through novel methodologies and algorithms for security policy generation, optimization of security policy implementations, and enhancement of the reconfigurability of the Security Status Monitor (SSM). We also establish a security policy database and assess the effectiveness of the extended framework for policy checking across various use case scenarios. Our experiments show that *EnSAFe* can detect runtime threats/vulnerabilities at low area overhead.**

***Index Terms*—SoC Security Monitoring, Upgradability, eFPGA.**

I. Introduction

Due to the continuous reduction in size and growing integration capabilities of contemporary semiconductor technologies, SoCs have become more extensive and intricate, featuring numerous intellectual properties (IPs) ranging from tens to hundreds. Furthermore, in order to enhance adaptability, versatility, and efficiency within the SoCs with dense and shrunk technology nodes, recent research indicates a keen interest in exploring the potential integration of moderately sized FPGA arrays, commonly referred to as eFPGAs, to serve a broad spectrum of applications. Schiavone *et al.* introduced the Arnold SoC [1], featuring an eFPGA designed for versatile, power-efficient IoT devices with energy constraints. This SoC demonstrates the utilization of the eFPGA as an extended subsystem within the SoC, effectively enhancing computational performance by offloading CPU tasks. Likewise, another study developed a reconfigurable SoC tailored for intelligent power applications [2]. This study illustrates that despite the eFPGA's notable increase in silicon area, it proves to be an outstanding solution in terms of both energy efficiency and reduced latency.

The application of eFPGA architectures extends beyond just enhancing performance and energy efficiency; recent research has also highlighted their usefulness in addressing security threats within the integrated circuit supply chain, such as reverse engineering, IP theft, and excessive IC production. P. Mohan *et al.* introduced a method called hardware redaction utilizing eFPGA to safeguard IPs [3]. Here, the eFPGA's bitstream serves as a secret protection against IP theft, overproduction of ICs, and reverse engineering attempts. An example illustrating the versatility of eFPGA usage is depicted in Fig. 1. In Fig. 1(a), the eFPGA within the Arnold SoC [1] is primarily employed for acceleration with a focus on energy efficiency. Conversely, in Fig. 1(b), the eFPGA is used to redact a security-critical IP/module, combating reverse engineering. In this scenario, performance enhancement is not a primary concern, so the eFPGA fabric consists solely of logic cells without DSP or memory components.

In the current landscape of the growing SoC market, the need for rapid time-to-market (TTM) is on the rise, leading to a higher prevalence of third-party IPs being reused. This practice introduces various unverified entities into the supply chain, consequently increasing the risk of potential vulnerabilities within the SoC architecture. Such vulnerabilities can have severe and possibly irreversible consequences [4]. As a result, the imperative for today's SoCs is the integration of a security monitoring and verification engine, a task that is becoming increasingly complex due to the escalating complexity of SoCs and shrinking time-to-market windows [5]. Notably, none of the existing eFPGA solutions have addressed this critical issue—how eFPGA can be harnessed to enhance security monitoring in modern SoCs, especially in real-world in-field operations (against zero-day attack[1]).

Over the past few years, there has been a significant surge in research efforts dedicated to exploring potential solutions for security monitoring, verification, and validation of SoC designs in order to protect them against known security threats and attacks [6]–[16]. Nevertheless, none of these solutions encompass the dynamic and adaptable nature of SoCs as part of their problem statement. Consequently, they all remain static

[1]For a newly discovered vulnerability, developing a solution and installing it often takes some time. In the meantime, an increasing number of attacks are performed to exploit this vulnerability, which is known as a zero-day attack.

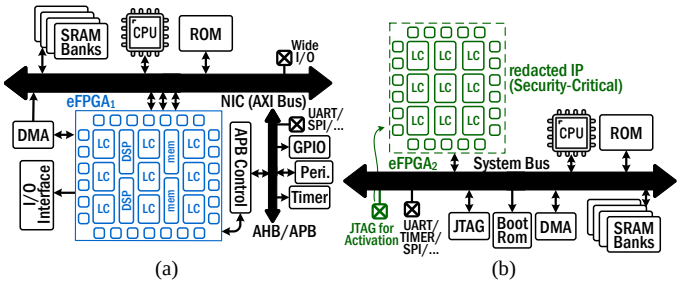


Fig. 1: The Engagement of eFPGA Fabric for (a) Acceleration and Energy Efficiency, (b) IP redaction against Reverse Engineering.

in nature, which renders them ineffective when it comes to countering zero-day and emerging attacks. To address this gap, we introduced *EnSAFe*, a reconfigurable security policy [17], [18] checking engine[2], in our previous work [19] that can be dynamically upgraded in the field. This capability ensures the sustainable protection of SoC designs against unforeseen and evolving threats. The foundation of the *EnSAFe* framework lies in the understanding that eFPGA fabrics are rarely utilized to their full capacity. In fact, various existing eFPGA automation tools, such as OpenFPGA [20], have reported notably low utilization ratios, particularly as designs scale in size. As depicted in Table I, a substantial portion of the eFPGA fabric remains unused, especially in larger designs. So, as shown in Fig. 2, *EnSAFe* empowers these automation tools to incorporate an independent and distinct security monitoring engine by repurposing the unutilized resources within the existing fabric, incurring minimal overhead. However, our previous work [19] lacked a structured approach for generating and implementing security policies. SSM generation and insertion was a manual process and thus inefficient. The SSMs also had limited reconfigurability and a simple architecture. Hence, we improve the *EnSAFe* framework in this paper to address these limitations. The main contributions of this paper are as follows:

(1) The extended *EnSAFe* framework is an eFPGA-based implementation flow, in which a regular eFPGA-based application (e.g., acceleration/efficiency) is merged with a security monitoring application at almost no cost (on the same fabric).

(2) We use an IP-specific yet SoC-level plug-and-play and configurable monitoring unit, termed as the security status monitor (SSM), for collecting (status of) security-critical implications. In this paper, we enhance the internal architecture and reconfigurability of the SSM.

(3) With SSMs, we present a reconfigurable and in-field upgradable security policy checking engine that will be mapped onto the unutilized eFPGA resources next to the main application.

(4) We devise an algorithm for security policy generation and formulate a security policy database based on common hardware vulnerabilities. We propose another algorithm for automated (through Python scripts) security policy optimization and SSM RTL code generation.

(5) We expand the scope of use case scenarios in this paper, presenting three additional scenarios targeting new vulnerabilities compared to our previous work [19]. This expansion not only enriches the evaluation but also demonstrates the versatility and effectiveness of the framework in addressing a broader range of security concerns.

TABLE I: eFPGA Utilization Ratio in OpenFPGA Framework [20]

Benchmark	# of Input Pins	# of Output Pins	LUTs Needed	eFPGA Size	Utilization Ratio (%)	Unutilized LUT
(4-64)-bit Counter	2	4-64	114	4×4	89%	14
Apex2	24-48	4	613	11×11	84%	163
SHA256	16-32	8-24	2845	20×20	88%	355
Elliptic Core	96-192	96-128	4064	28×28	64%	2208

II. Prior Art: SoC-level Security Monitoring

Over the past two decades, there has been remarkable progress in the development of hardware-oriented security solutions and countermeasures. Nevertheless, a significant proportion of these advancements have primarily concentrated on security countermeasures at the IP-level. Examples include techniques like hardware obfuscation [21], logic locking [22], [23], watermarking [24], [25], camouflaging [26], and side-channel analysis [27]. These methods, however, are not suitable for addressing vulnerabilities at the SoC-level, as many of these vulnerabilities stem from issues that span across multiple IPs or components [28]. Recent studies, however, show a shift to presenting SoC-level solutions, and in the case of SoC-level monitoring[3], following are the notable ones:

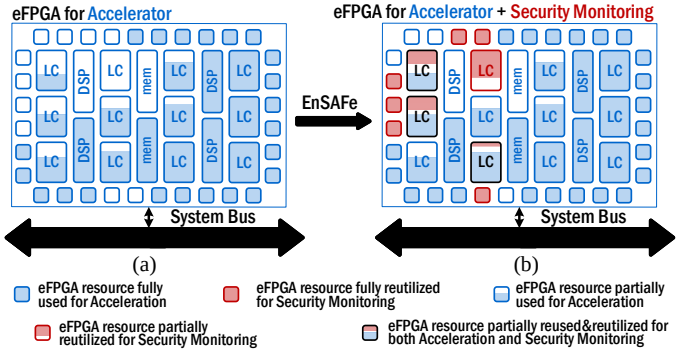


Fig. 2: Re-use of Unutilized Resource for (Decoupled) Security Monitoring.

Some studies, e.g., [6], [7], [12], [29], attempt to implement a centralized control engine for implementing SoC security policies. A centralized on-chip module was proposed in [6] which contains two FSMs for security configurations. On the other hand, [7] introduced a firmware-upgradable central controller for policy implementations. However, both [6] and [7] use IEEE P1500 [30] wrapper to communicate with other IPs. Another study, [12], proposed hardware and software IP management modules where each IP has its own hardware management module. Software management module acts as the central enforcer and it communicates with the SELinux policy server. The studies [6], [7], [12] have limited configurability either through firmware or through scan chain but they fail to offer in-field upgradability of policy implementations after deployment. Additionally, the types of policies covered in this breed require heavy intra-IP information with expert knowledge, which is rarely the case. Interestingly, the study [29] uses a reconfigurable hardware which is upgradable in-field. But the reconfigurable hardware was only used for security primitive implementations while a set of hardware monitors were used for vulnerability detection.

Similarly, few studies, e.g., [8], [9], follow design-for-debug (DfD) infrastructure to monitor intra-IP signals and forward events to a centralized policy engine to detect SoC violations. However, relying on DfD to observe the internal signals requires addressing many challenges (e.g., Re-purposing the DfD should not interfere with debug usage of the system). Further, to adapt to this DfD-security architecture, the IP provider needs to map security-critical events to the DfD instrumentation for the IP, which is a very hard assumption in modern SoCs with numerous 3PIPs. Some other studies, e.g., [10], [14], use a set of distributed monitors for bus monitoring but they do not have any defined central monitors. These distributed monitors are connected with the processor core and they are configurable through firmware upgrade. But the reconfigurability option is very limited in these studies. Few other studies, e.g., [15], [16], focus on firmware-upgradable hardware monitors for the

[2] *EnSAFe* is designed and dedicated for only eFPGA-equipped SoCs, whose eFPGA is customized for either acceleration or energy efficiency.

[3] We only focus on SoC-level monitoring prior works as the main focus of this paper is to introduce an SoC-level security monitoring engine.

processor. These hardware monitors are used to implement software execution-related security policies. As the monitors are highly customized for the processor, they are not suitable for bus monitoring and SoC-level vulnerability detection.

Lyu *et al.* identified common SoC security vulnerabilities by analyzing the design [13]. To monitor these vulnerabilities, the authors defined several classes of assertions and inserted them into the SoC design to enable runtime checking of security vulnerabilities. Another work in [11] proposes a high-level security policy language and compiler for policy synthesis. The synthesized policies are inserted into the SoC as hardware runtime monitors. The assertions [13] and synthesized policies [11], however, are for static systems and cannot be updated in-field, leaving the SoC susceptible to emerging security threats.

III. Threat Model

The security monitoring threat model within an SoC involves the identification and mitigation of potential security threats that could jeopardize confidentiality, integrity, and availability [31]. Preserving confidentiality involves safeguarding sensitive information within a design against unauthorized access and potential data leaks. Integrity pertains to the consistency and dependability of a design's functionality over time. Availability signifies the continuous presence of essential information within the design, regardless of circumstances. The threat models encompass the subsequent scenarios:

(1) Unauthorized Access: An attacker may gain access to the system or device by exploiting vulnerabilities in the hardware or software. They can also try to gain access through brute-force attacks or exploit the debug interface. Unintentional design mistakes, bugs, or improper authentication mechanisms may create exploitable entryways for the attacker.

(2) Malware: Attackers can execute malicious software codes on the device, which can be used to access secret data or disrupt normal operations. Ransomware is an example of malware that encrypts data and demands payment in exchange for the decryption key. An attacker may also try to bypass the privilege control and gain access to the kernel mode.

(3) Tampering and Physical Attacks: Attackers may physically tamper with the hardware, e.g., through fault injection or side-channel attacks. In this case, the attacker has physical access to the chip and they modify the hardware or firmware to malfunction the device or to analyze the physical characteristics.

(4) Supply Chain Attacks: Supply chain attacks involve compromising the hardware or software at some point during the manufacturing or distribution process before the product reaches the end user. This can be used to introduce backdoors or other malicious components into the device or system.

(5) Information Leakage: An attacker may try to leak sensitive data from a secure location to external ports of the device. Attackers with physical access to a device or system can extract sensitive information by accessing the device's storage or memory, or by intercepting the device's system bus.

(6) Denial of Service (DoS) Attacks: Attackers may try to halt a security operation (e.g., device authentication) or cause the system to stop at a vulnerable condition. This may also cause the system to become non-operational for a certain time.

To address these threats, SoC security monitoring involves implementing a range of security measures, including distributed monitoring units and real-time analysis capability. The security monitoring approach also needs to be upgradable to detect and address unforeseen zero-day attacks, which can be a completely new attack or a new form of a known attack.

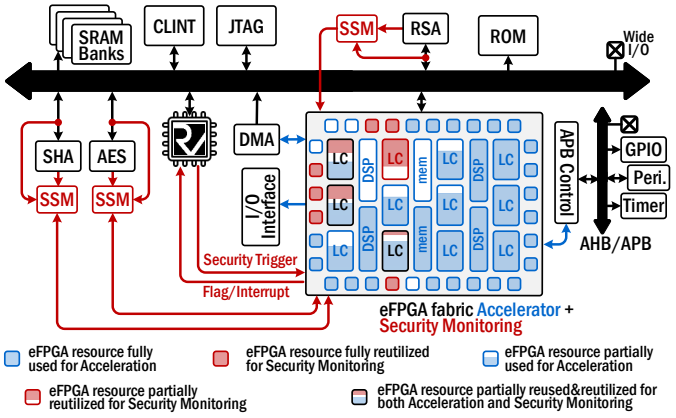


Fig. 3: Security Monitoring via the eFPGA and Security Status Monitor.

IV. Proposed Scheme: EnSAFe

As shown in Table I, when the circuit size targeted to be mapped into an eFPGA fabric is getting larger, the fabric size will explode, and the utilization ratio decreases significantly. Further, as the utilization of larger eFPGA fabrics decreases, the number of resources available (unutilized) increases. For instance, in a 28×28 eFPGA fabric, there are more than 2K LUTs unutilized, while only 14 LUTs are unutilized in a 4×4 eFPGA fabric. Since eFPGAs will be mostly used for acceleration purposes (e.g., to model a complex compute-intensive kernel migrated from the CPU to the eFPGA), which requires a large fabric, it opens the possibility of reutilizing unused resources for another (non-computation-intensive) application. So, *EnSAFe* enables the security monitoring of the entire system in a dynamic manner on these unutilized resources (Fig. 2), which is typically not a resource-intensive application and can be fit on the unutilized part. Fig. 3 demonstrates the outcome of *EnSAFe* in a RISC-V-based SoC architecture. As shown, it consists of two main components: (1) eFPGA for acceleration/security, which realizes both acceleration and dynamic monitoring together, and (2) security status monitor (SSM), which is responsible for collecting required security-critical implications in a compressed manner. Enabling the security monitoring on eFPGA fabric allows us to upgrade the security policies in-field at runtime by reprogramming the eFPGA, which helps address new vulnerabilities or to prevent zero-day attacks.

A. eFPGA for Acceleration+Security

In this setup, the eFPGA connects to the system bus for hardware acceleration and directly monitors bus transactions. It allows the CPU to configure the eFPGA, using configuration bits as triggers for monitoring based on specific policies. This capability facilitates direct system bus observability, enabling easy detection of vulnerabilities that affect bus transactions. These policies, implemented within the eFPGA, analyze data from both the system bus and the SSM (refer to Subsection IV-B) to identify security vulnerabilities. Given the dual use of the eFPGA for both acceleration and security, there's a limit to how many security policies it can accommodate, necessitating policy ranking and optimization (as detailed in Algorithm 2). *EnSAFe* employs enforced memory mapping to secure eFPGA access against unauthorized IPs throughout SoC lifecycle [19][4].

[4]Since multiple components may be authorized to engage eFPGA for applications, *EnSAFe* regulates them by defining specific security policies.

Using eFPGA fabric for acceleration and security monitoring has numerous benefits. One significant advantage is that it allows for the efficient use of hardware resources. When eFPGA fabric is utilized for acceleration purposes, the unused resources can be repurposed for security monitoring. Our investigations show that the unutilized parts in the eFPGA are reasonably big (shown in Table I) and our experiments in Section VI will present how security policies can be mapped using these resources. Thus, the system designers can maximize the use of hardware resources, reducing waste and increasing efficiency. Additionally, eFPGA fabric for security monitoring can provide an added layer of protection for systems. By monitoring the security of a system in a dynamic manner, *EnSAFe* can help detect and prevent security breaches, protecting sensitive data and systems from unauthorized access or malicious attacks.

B. Security Status Monitor (SSM)

The eFPGA acts as the core monitoring unit in our approach but it has limited observability due to access to only the system bus. Security-critical IPs and registers are distributed across the design and vulnerabilities can occur at runtime in any of these IPs. Also, not all of these security vulnerabilities may have an observable impact on the system bus transactions. Thus, observability of these security-critical instances and registers is a must for runtime detection of security vulnerabilities. Hence, we propose a new monitoring module, called SSM. The SSMs, as shown in Fig. 3, are added only for IP-internal (security-critical ones) monitoring, and they follow a very generic definition to minimize the IPs' customization[5].

The SSM concentrates mostly on inter-IP transactions. The SSMs will also take care of a limited amount of intra-IP signaling, which is difficult to achieve by only observing the system bus transactions. Based on the functionality, timing, and I/O connections of the IP, they filter out and store critical data of interest. Then, this data will be sent to the eFPGA (if triggered for monitoring). SSMs are a static part of the SoC and all the security policies in *EnSAFe* are written based on the information available through the SSMs (and main bus). Although it seems that having SSMs static may limit the reconfigurability, we designed the SSMs with configuration options for event detections. In *EnSAFe*, the eFPGA sends configuration bits to these SSMs to properly configure them. Our case studies show these SSMs increase the observability of *EnSAFe* and thus can cover a wide range of vulnerabilities.

Fig. 4 shows an overall view of the SSM architecture. Each SSM is connected to the eFPGA fabric directly[6] through dedicated MMAP WR and RD channel as shown in Fig. 3. In both channels, the address signals are 10-bit wide, and the data signals have a width of 32 bits. The security policy within the eFPGA sends configuration bits (32-bit) to configure the parameters of the SSM or to enable data monitoring for an event using the MMAP WR channel. An encoder logic, within the MMAP WR channel, transmits the configuration bits and trigger bits (sent from the eFPGA) to the SSM controller. Similarly, a decoder logic, incorporated within the MMAP RD channel, is used to send important data from the SSMs to the eFPGA. The security policies implemented within the

[5]SSMs resemble performance monitors used in today's modern SoCs [32].

[6]As there will be multiple SSMs in the SoC, each for a security-critical IP, the communication between the eFPGA and the SSMs through a shared private bus may create a bottleneck. Since eFPGA for acceleration occupies a significant portion of die size, our experiments show slight delay for routing dedicated wires between IPs' SSMs and eFPGA.

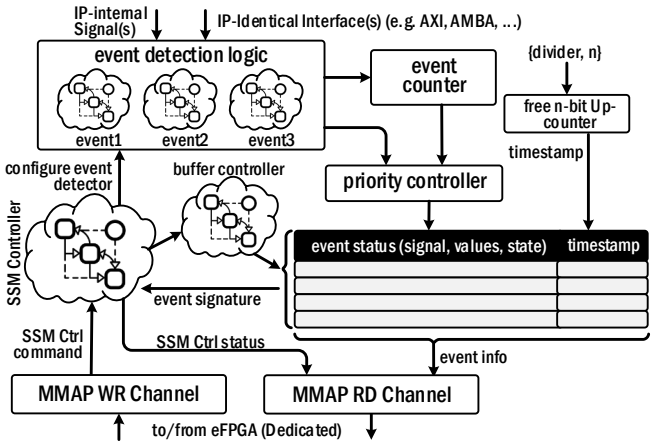


Fig. 4: Architecture of a Security Status Monitor.

eFPGA use the event information from the SSMs to detect security vulnerabilities. The encoder logic translates received data from the eFPGA into a specific format of configuration bits (shown in Fig. 4 as SSM Ctrl command) required by the SSM controller logic. The configuration bits may contain details (e.g., enable/disable settings, register initialization value, parameter for event detection, etc.) regarding the security events to be checked, buffer control bits, or priority settings. The SSM controller decodes each part of the configuration bits and programs the configurable registers within the event detection, buffer controller, and priority controller modules.

The event detection logic is the most important module of an SSM. We design SSMs to have an identical connection to IPs (as IPs use the same handshaking, e.g., AXI) along with a set of IP-specific connections/signals for monitoring. The event detection logic reads directly from these IP-specific connections or signals as shown in Fig. 4. This detection module consists of multiple FSMs dedicated to identifying one or more events. It comprises configurable registers, comparators, and tiny FSMs. The internal logic (combinational or sequential) for each event detection is crafted based on whether the event occurs within a single clock cycle or spans multiple cycles. The events are defined based on the I/O values of IP-specific signals and interface signals. For our implementation, we assign an integer ID to each of these events that we monitor. The event detection logic looks for these events or event sequences in the I/O signal values (address, data, valid, ready, etc.) using the comparator logic. This module is designed to be configurable, allowing the SSM controller to modify its configuration. This adaptability is achieved through the integration of a separate FSM within the event detection module, responsible for updating register contents based on inputs from the SSM controller.

The SSM features an up counter (event counter in Fig. 4) that monitors the events and halts based on predetermined conditions. For example, if the counter is monitoring an AES process, it may stop after a certain number of cycles required to finalize the process. The counter may also pause when the AES module asserts the $Done$ signal. After the counting process is complete, a status register will save all the essential signal values. The outputs from the event detection logic and the event counter go into a priority controller. We used the priority controller module to set the priorities among multiple detected events. In cases where multiple events are detected in a single clock cycle, this module helps to determine which event data should be promptly forwarded as output to the next module. By

TABLE II: Sample Security Policy Database

Security Policy Database	Security Policy Description	Security Violation	Ramification
Debug related	During the HALT operation, the value of the Program Counter should remain unchanged	Integrity, confidentiality	Unauthorized access control
	Target and host processor should be in the same privilege level during inter-processor debug session	Confidentiality, integrity	Unauthorized access control
	The assets of an SoC should not be transferred using the system bus during the HALT operation	Confidentiality, integrity	Unauthorized access control
	Every debug session should be protected with a password, and the password checking bit should be de-asserted during reset	Confidentiality, integrity	Unauthorized access control
Cryptographic IP related	Any part of the cipher key should not be leaked through the primary output port	Confidentiality	Information leakage
	The ciphertext should always be cleared during reset	Confidentiality	Denial of service (DoS)
	The DONE signal of any crypto operation should be asserted after the final round of the process	Integrity	Information leakage
	The BUSY and DONE signal for any crypto module should not be asserted at the same time	Integrity	Information leakage
Memory access related	Reading or writing in the same memory bank should not cross a certain threshold during a fixed amount of time	Confidentiality	Illegal memory access
	Any untrusted IP controller should not get access to the protected memory region	Confidentiality	Illegal memory access
	Every IP inside an SoC should have a well-defined and non-overlapping memory region in the system address map	Confidentiality	Illegal memory access
Processor related	The privilege level for every important control and status registers inside the processor should be fixed for a single instruction execution	confidentiality	Unauthorized access control
	The privilege level of the processor should be restored to M-mode while returning from a debug operation	Integrity	Unauthorized access control
	The processor of an SoC should monitor and record any instances of hardware exceptions	Integrity	Unauthorized access control

default, all security events are assigned equal priority. In cases of conflict, where multiple events share the same priority, we resolve the conflict by allowing the event with the lower event ID to proceed first. However, in some scenarios, we may want to prioritize the detection of a security violation over others, and this can be achieved by configuring the priority settings in the priority controller. However, this brings up a trade-off as we may lose the chance of capturing vulnerabilities from the low-priority IPs at specific periods of time. We tried to minimize data loss during conflicts by storing low-priority data in temporary registers. However, we intentionally limited the number of temporary registers to twice the number of events to keep the resource utilization and design complexity of the module low.

We also have a limited number of buffers to temporarily store security event information/data based on the output from the priority controller. The buffer stack utilizes a FIFO (first-in, first-out) approach to hold the data for a temporary period until they are read out by the eFPGA. In addition, we have a buffer controller FSM to control the buffer read/write functionality. At design time, the depth of the FIFO can be adjusted within the buffer controller module depending on the number of events being captured by the SSM. In our experiments, we adjusted the FIFO depth (16$\sim$32) based on the IP functionality and events being monitored so that we do not miss any critical data due to buffer overflow. We used another free up-counter to build the timestamp for recording the events. The SSM monitors and collects data with timestamps so that the eFPGA will be able to tell at which time the vulnerability occurred. The current timestamp value is tied to each event being stored in the buffer stack. If the event detection logic requires comparisons with past values of an event stored in the buffer stack, the SSM controller logic retrieves the event status from the buffer, depicted as the event signature in Fig. 4, and forwards it to the event detection logic. The SSM controller filters required event information from the buffer based on the event ID and timestamp comparison. As mentioned before, the policy within the eFPGA will read the event data from the buffer stack (shown as event info) through the MMAP RD channel. The policy may also read SSM control status (e.g., event detection enable/disable status, FIFO depth parameter, priority configuration, etc.) using the RD channel.

C. Security Policy Database

To ensure effective security monitoring, it is crucial to establish a strong set of security policies. This is particularly important given the increasing complexity of contemporary SoCs, as emphasized in Section I. The creation of a comprehensive security policy database is essential and involves using various open-source architectures, analyzing important security-critical IPs, and assessing different attack strategies [33]–[35]. Algorithm 1 outlines the steps involved in developing a set of security policies for an SoC. The process starts with acquiring detailed knowledge of the SoC, obtained either through the examination of RTL codes or the full specification of the SoC (line 1). Additionally, a background understanding of common hardware attacks [33], [36], [37] is imperative to identify the attack points for that SoC (e.g., side-channel, fault-injection, tampering, trojan, bus snooping, etc.) (line 1). When developing the security policies shown in Table II, we had access to the RTL codes of the Ariane SoC. We examined the Ariane SoC architecture to understand the bus protocol, memory hierarchy, IP functionalities and responsibilities, etc. We also explored common hardware vulnerabilities associated with IPs (e.g., AES, DMA, RISC-V processor, etc.) and functionalities (e.g., debug, malware, runtime, etc.) from [33], [36], [37]. Next, we proceed to analyze the signals and IPs of the SoC for each attack point (lines 2 to 3). This involves a detailed examination of I/O ports, bus signals, and peripheral connections to pinpoint security-critical transactions (line 2). A transaction is considered security-critical if it involves the transfer or handling of an asset. Similarly, an in-depth analysis of security-critical and susceptible IPs, including cryptographic IPs, memory controllers, and bus controllers, is conducted (line 3). We then identify possible attack surfaces for the SoC under consideration (line 4). For our use case scenarios, discussed in Section V-B, using Ariane SoC, we finalized four attack surfaces: I/O interface, communication bus, peripheral devices, and memory subsystem. We then use our analysis results, combined with the knowledge of attack points and SoC architecture, to identify vulnerabilities (line 5). For instance, in generating the sample security policy database presented in Table II, we

Algorithm 1: Security Policy Generation

Input: RTL/Specification of SoC design (D), Critical attack points (AP);
Output: Security Policy Database ($\mathbb{P}$);
1 **for** *each attack point* $AP_i \in AP$ **do**
2 ST= Signal_Transaction_Analysis(D);
3 IP= Security-critical_IP_Analysis(D, ST);
4 AS= Attack_Surface_Analysis(AP, D);
5 $\mathbb{V}_{ID}$= Vulnerability_Idenfication(AP, D, ST, IP, AS);
6 **for** *each vulnerability* V_i *in* $\mathbb{V}_{ID}$ **do**
7 V_f= Vulnerability_Categorization(V_i, AP, D);
8 Asset, $\mathbb{A}$= {};
9 $\mathbb{A}$= $\mathbb{A} \cup$ New_Asset_Identification(V_i, AP, D);
10 T= Threat_Model_Identification($V_i, \mathbb{A}, AP, D$);
11 $\mathbb{P}$ = $\mathbb{P} \cup$ Security_Policy_Formation($V_i, T, \mathbb{A}, AP, D$)
12 **return** $\mathbb{P}$;

identified vulnerabilities such as access control, information leakage, denial of service, and illegal memory access. We then proceed to the next step where we generate security policies for each identified vulnerability (lines 6 to 11). We first categorize similar vulnerabilities into groups to facilitate the identification of assets (lines 7 to 9). An asset refers to any valuable data or information that needs to be protected from unauthorized access, manipulation, or compromise [38]. We examine the groups of vulnerabilities and identify the assets they are trying to access, manipulate, or compromise. After identifying the primary and secondary assets [38] for the vulnerability, we proceed to the next step and define the threat model (line 10). A threat model typically describes several components including the assets, attack surface, threat actors (who can attack and their capabilities), and vulnerabilities. A brief discussion on the threat model for the Ariane SoC is provided in Section III. Finally, we formulate security policies for the threat model (line 11). We define security policies as a set of rules and guidelines that define how the assets will be protected against security threats [17], [18]. Based on the IP types and design functionality, we have classified the security policies into the following categories:

(1) Cryptographic IP Related: Cryptographic IPs in a SoC handle security assets and pose a significant security risk. This risk can lead to breaches of security objectives, like confidentiality violations if a hardware Trojan is activated in the AES engine, revealing the cipher key. Another Trojan in the AES engine can disrupt encryption/decryption, compromising system availability. Therefore, robust security policies for IPs like AES, RSA, or SHA are vital to protect sensitive assets. Some of the sample security policies can be:

- AES engine should assert a $DONE$ signal after the completion of the final round of the operation, not after any intermediate rounds.
- The cipher key for any crypto IP should not be leaked through the primary output.

(2) Memory Access Related: In an SoC with memory dedicated to different security-critical IPs, an access policy would be defined so that critical memory locations are inaccessible from untrusted third-party IPs. However, security bugs like hardware trojans can create backdoors, allowing illegal memory access for untrusted IPs. Threats like Rowhammer exploit frequent memory read/write operations. Implementing security policies, such as limiting read/write access to specific timeframes, is crucial to prevent unauthorized memory access and defend against Rowhammer attacks.

(3) Processor Related: Software exploitable hardware vulnerabilities, i.e., cross-layer attacks, can present an ongoing danger to the processors and their internal execution of instructions. These bugs are difficult to identify and may avoid security

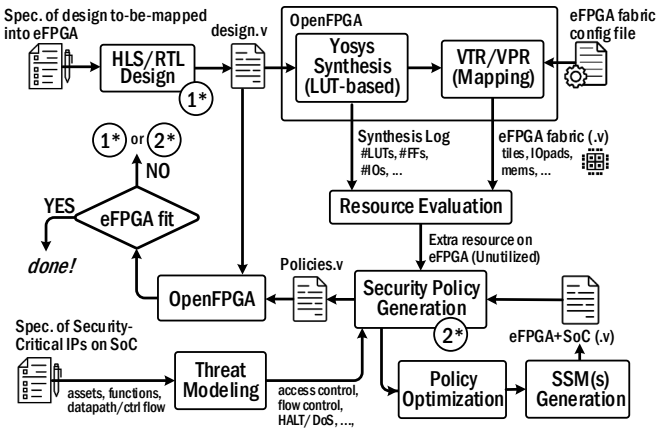


Fig. 5: Overall Flow of the *EnSAFe* Framework.

checks. Consequently, it is essential to implement security policies that safeguard proper privilege permissions for various low-privileged software operating within the system. For instance, when conducting a debug session, the processor can remain at any privilege level (typically at the highest privilege level), but when concluding a debug session, the processor must restore the highest privilege level.

(4) Debug Related: Debug module is typically used for reading and modifying processor internal registers and peripherals. Debugging includes halting of execution of any instruction when the processor raises an exception. This halt operation can be initiated by an off-chip debugger from a host processor via the Debug Access Port (DAP). If a debug session is compromised, it could have severe security consequences if not carefully monitored. To prevent unauthorized access via the DAP during processor operation, security policies have been devised based on various debugging scenarios. For example, one important policy can be that the assets of an SoC should not be transferred using the system bus during the HALT operation.

D. Overall Flow of the EnSAFe *Framework*

Fig. 5 shows the overall flow of the *EnSAFe* framework. As the eFPGA fabric will be used for both accelerator application and security policy implementations, the flow starts with mapping the accelerator design on the eFPGA fabric. The HLS/RTL implementation of the accelerator design is used to generate the eFPGA fabric netlists. As shown in Fig. 5, after generating the eFPGA fabric for the accelerator (based on the targeted eFPGA architecture), resource evaluation has been done to check the availability of unutilized resources. We map targeted security policies on these unutilized resources. Next, based on the availability of unutilized resources, security-critical IP specifications, and potential threat models, we create a security policy database and implement these policies in RTL. The policies are created in such a way that they can cover almost all of the known possible attacks under the threat model. To make the most efficient use of the eFPGA fabric, the policies need to be optimized and implemented in such a way that they will have maximum overlapping hardware resources. Thus, our optimization strategy involves looking for overlaps between different policies and merging those policies so that the implementations are resource and latency optimized.

Algorithm 2 shows the general steps for optimizing policies and creating SSM RTL codes. After we implemented the security policies in RTL (line 1), we used several Python scripts to optimize the policy implementations and create SSM

template codes. These Python scripts follow the steps outlined in Algorithm 2. At first, we analyze each security policy implementation and categorize them into groups with common I/O ports (lines 1 to 5). The Python script responsible for these steps reads each policy RTL code and creates a dictionary with the input-output ports (line 2). We have extensively used PyVerilog [39] tool in our scripts to parse, analyze, and generate RTL codes. Subsequently, the dictionary is sorted and uniquified with respect to input-output ports. Thus, we create groups of policies with common I/O ports (line 3). Next, we create another Python dictionary with the groups as "keys" and the parsed RTL codes (using PyVerilog) as values (line 4). We also create Python lists of I/O ports from all policies in a group (line 5). Proceeding to the next step, we optimize each group of policies (lines 6 to 12). The Python script responsible for these steps reads the Python lists of I/O ports from line 5. Leveraging the Code Generator module of the PyVerilog tool [39], we create RTL wrapper codes that incorporate the list of I/O ports for each group of policies (line 7). Next, we need to analyze the combinational and sequential logic of the policy implementations. This is accomplished using the Dataflow and Control-flow analyzer of PyVerilog tool [39] (line 8). We then combine the RTL logic for each group of policies and write them into the RTL wrapper code using PyVerilog Code Generator (line 9). We then modify the parsed AST (Abstract Syntax Tree), generated by the Code Parser, to create interrupt outputs and to add necessary custom logic, signals, and port mapping (lines 10 to 11). The modified AST is then forwarded to the Code Generator module, and the generated RTL code is stored as the optimized implementation for each group of policies (line 12). The third Python script works on the optimized policy implementations from line 12 and creates SSM wrapper code (lines 13 to 21). We utilize the Code Parser module to read each optimized policy RTL and analyze the input ports of the policy (line 14). We also create Python lists with the input ports. Following this, a Python dictionary is created with the names of the system bus, private bus, and IPs of the Ariane SoC serving as "keys". Using the Dataflow analyzer of PyVerilog, the driver source (e.g., system bus, private bus, and IPs) of each input port in the Python list is determined. The input port is then put as a value into the Python dictionary with the corresponding driver source matching a key. Consequently, groups of input ports are formed based on their driver source (line 15). Next, for each group of input ports, we check whether the eFPGA has observability into them. This involves comparing the input ports with the list of signals where the eFPGA has direct observability. If the comparison fails and the eFPGA has no observability into the input ports, we proceed to generate the SSM template code (line 17). We first use the Code Generator module to create an RTL wrapper with the input ports intended for monitoring with the SSM (line 18). We then modify the parsed AST to add MMAP (memory-mapped) I/O read-write logic and include a configurable buffer, priority controller, and counter logic (lines 19 to 20). The modified AST is then forwarded to the Code Generator module, and the generated RTL code is stored as the SSM RTL code (line 21). The generated SSM RTL code serves as a template code that needs further customization, particularly for the event detection logic and the SSM controller logic.

Security policies in the *EnSAFe* framework are categorized based on their monitoring needs: some require system bus monitoring, while others need IP-internal monitoring. Due to eFPGA resource constraints, we prioritize and shortlist policies for implementation. For IP-internal monitoring, we generate SSM sub-circuits and integrate them into the SoC. At the same time, we map policy sub-circuits onto the eFPGA fabric with accelerators. If the eFPGA tool (VTR/VPR) fails in remapping, we have two options: (1) redesign the accelerator (prioritizing security) or (2) select fewer policies (prioritizing performance). However, no remapping failures have occurred in our case studies so far.

Algorithm 2: Policy optimization and SSM generation algorithm

Input: RTL codes of N security policies ($\{P_1, P_2, P_3, ..., P_N\}$);
Output: Optimized RTL codes $\{O_1, O_2, O_3, ..., O_M\}$ of M security policies; RTL implementation of S SSMs;

1 **for** *each of the N policies* **do**
2 Analyze the I/O ports of the policy;
3 Create M groups of policies with common I/O ports, $M_i = \sum_{j=1}^{N_k} P_j$, where $P_j, P_{j+1}, ..., P_k$ have I/O ports in common and $\sum_k N_k = N$;
4 Put the policy in the corresponding group, $P_j \in M_i$;
5 Create a set of I/O ports from all policies in group M_i; $U_i = \bigcup_{j=1}^{k} U_k$;
6 **for** *each group of policies (M_i)* **do**
7 Create RTL wrapper code with U_i I/O ports of the group;
8 Analyze the combinational (L_{ci}) and sequential (L_{si}) logic of each policy (P_i) in M_i. Here, $\{L_{ci}, L_{si}\} = \sum_n L_{ni}$ = all logic blocks/statements in P_i;
9 Combine RTL logic of the policy implementations ($\bigcup_{i=1}^{m} \{L_{ci}, L_{si}\}$, where m is the total no. of logic in group M_i);
10 Create N_k-bit interrupt outputs (1-bit interrupt for each policy);
11 Put necessary custom logic, create signals and port mapping of interrupts;
12 Store the RTL wrapper as the optimized implementation (O_i);
13 **for** *each of the optimized policies (O_i)* **do**
14 Analyze the input ports I_j of the policies;
15 Categorize input ports into S groups based on sources of signals (e.g. system bus, private bus, IPs, etc.), $I_j \in S_i$;
16 **for** *each S_i group of input ports* **do**
17 **if** *eFPGA does not have observability into I_i* **then**
18 Create RTL wrapper code with I_i input ports for group S_i;
19 Create MMAP I/O ports and logic for the read and write channel;
20 Put customizable buffer, priority controller, and counter logic into the RTL wrapper;
21 Store the RTL wrapper as the SSM RTL code, S_k;
22 ***return*** optimized implementations ($\{O_1, O_2, O_3, ..., O_M\}$), SSM RTL codes ($\{S_1, S_2, ..., S_S\}$);

Once the security policies are mapped into the eFPGA and the SSMs are connected for collecting data, in case of any security policy violation, either the eFPGA or the CPU can take care of the policy enforcement. However, to keep resource utilization for the security portion at a minimum, we have considered the CPU as the enforcement entity. So, as shown in Fig. 3, depending on the nature of the policy, the eFPGA raises a flag or sends interrupts to the CPU. The CPU then calls the ISR (Interrupt Service Routine) written for the interrupt bit, which may in turn read the policy status register within the eFPGA using the system bus. The policy status register contains policy-specific data (i.e., memory address, bus master ID, IP output data, etc.) collected/derived from the SSM. The ISR then takes any necessary actions from there, depending on the type of violation. We will discuss the CPU enforcement actions in more detail in Section VI-A.

V. Case Studies

A. Experimental Setup

To assess the performance and efficiency of the extended *EnSAFe* framework, we targeted Ariane SoC [40] to be enabled with both acceleration and security monitoring. Ariane SoC consists of a 64-bit, 6-stage, in-order RISC-V processor and

several (peripheral) IPs like UART, SPI, Ethernet, GPIO, etc., all connected using AXI crossbar. We augmented the SoC by adding a 128-bit Advanced Encryption Standard (AES) crypto module and a 64-bit Direct Memory Access (DMA) module. Additionally, to have a fair comparison, for the acceleration, we inserted an eFPGA with approximately equal size to the eFPGA used in Arnold SoC [1]. For building the eFPGA for the integration, we engaged OpenFPGA framework [20], which provides a complete Verilog to bitstream generation flow with customizable eFPGA architectures (suited for the design to be mapped). Using the framework summarily shown in Fig. 6, the OpenFPGA framework creates a tightly organized set of programmable tiles, block RAMs, DSP blocks (if needed), and I/O banks. We integrated the generated eFPGA fabric into the Ariane SoC through the system AXI4 crossbar. Additionally, for SSMs, direct connections to (unutilized) I/O banks are added. For overhead comparison, all experiments went through Synopsys Design Compiler using open Skywater 130nm process [41]. Also, the test of the security monitoring scenarios has been done using Vivado (for synthesis and integration) and OpenOCD (JTAG-enabled debugging). Further, the test are accomplished (with functional verification) using an emulation model on Genesys 2 Kintex-7 FPGA Development Board [42].

B. Use Case Scenario

We have considered six use case scenarios for eFPGA-based security monitoring proof of concept. They involve six different vulnerabilities and their detection approach using the proposed eFPGA-based monitoring architecture. Table III lists and describes these targeted vulnerabilities.

(Vulnerability 1) Rowhammer Attack: Rowhammer attack [43] is performed by repeated read/write and recharge of a DRAM row. This attack targets the security assets that have been stored in DRAM banks during the execution of different applications on SoC. The repeat of read/write will redirect access to different rows of the DRAM on the same bank, allowing the adversary to catch values not requested for (illegal access).

To model this attack, we defined the security policy as: **repeated read/write accesses should not exceed a certain threshold within a limited time**. In this case, the security policy will monitor DRAM bus continuously to find high-frequency read/write (count of read/write + timestamp). The security policy implementation can look for the rowhammer attack simultaneously in multiple banks (up to 8 banks), and it is parameterized for the number of memory banks, threshold value for counting, and time window (clock cycles). The threshold value corresponds to a physical attribute of the DRAM which defines the minimum frequency of row buffer recharge that may result in rowhammer effect. We experimentally chose this value for the DRAM during policy implementation. However, the threshold value does not have any cost impacts on the policy implementation. In the following snippet, the security policy for rowhammer detection is formulated at a high level.

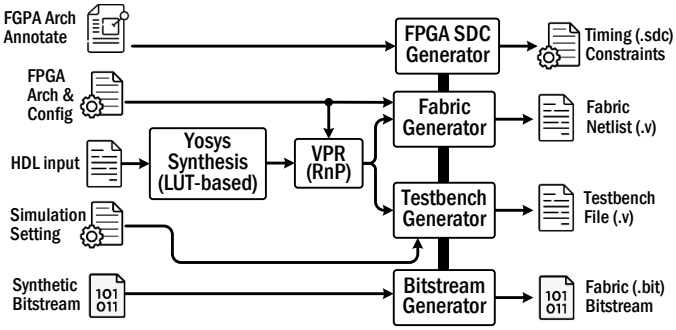


Fig. 6: Custom eFPGA design flow based on OpenFPGA [20].

```
if time_window_start
    count = 0;
else if read/write_addr_within_bank
    count = count + 1;
if count > threshold
    interrupt = 1;
```

(Vulnerability 2) DRAM Access Control: DRAM access control violations occur when an unauthorized party gains access to the memory regions that they are not supposed to access. Various reasons can lead to access control violations, including hardware bugs, software bugs, and malicious attacks. In this case, an unauthorized party can read, modify, or delete sensitive information stored in DRAM, including passwords and encryption keys. In order to prevent DRAM access control violations, computer systems employ a variety of security mechanisms. One such mechanism is memory protection, which ensures that memory locations are only accessible by authorized parties. This is done through access control lists enforced at boot-time, which specify what IPs or processes are allowed to access which memory locations. However, improperly established memory access policies, especially for aliased memory regions, can create an entry point for attackers [CWE-1257]. Another example of DRAM access control violation is DMA attacks. In this case, certain IPs, referred to as early boot IPs, may possess DMA capability and may power up before the boot process is complete [CWE-1190]. If these IPs are not trusted, they could potentially allow an attacker to carry out DMA attacks to gain access to protected assets.

The security policy for this access control vulnerability monitors the DRAM read/write requests coming from each bus master (including each DMA-capable IP) and validates these requests against the memory access control policy for that IP. A very high-level representation of this policy is shown below.

```
if read/write_addr_within_secure_region
    identify_request_source;
    check_access_policy_for_request;
    if request_violates_policy
        interrupt = 1;
```

(Vulnerability 3) Illegal Execution of Cache Flush Instruction: Typically, the *MSTATUS* register within a RISCV-based design holds crucial signals that control the privilege information for the operating systems in the processor. The cache flushing instruction (*SFENCE.VMA*) is usually permitted to run only in the highest privilege level, which is M mode, in any host OS built on RISCV design. However, any guest OS, such as a virtual machine connected to the system, can execute any instruction in the Supervisor (S) mode or the User (U) mode. It is essential to note that caches may contain sensitive security information and therefore must not be flushed from any other privilege level except the highest one (M mode). Unauthorized users may also exploit cache flush instruction execution to perform other attacks like Rowhammer. The Trap Virtual Memory (*TVM*) bit within the *MSTATUS* register regulates whether the *SFENCE.VMA* instruction can be executed from lower privilege levels or not. If the *TVM* bit is set, the instruction's execution in the lower privilege levels will lead to a hardware exception. On the other hand, if the *TVM* bit is cleared, the guest OS can execute the flushing instruction from lower privilege levels, which could result in a possible access control violation. This bit can be altered either by designers' mistakes or by intentionally inserting a bug.

TABLE III: Vulnerabilities Targeted for Monitoring in *EnSAFe*

Vulnerability	Description	Category	Security Objective	Security Policy Monitor	SSM Event	FPGA-based Resource Needed for the Policy
Rowhammer Attack	Repeated reading/writing and recharging of a DRAM row for illegal access to security asset(s) stored in RAM	CWE-1260	Confidentiality	Repeated rd/wr req should not exceed a threshold within a limited time	N/A	61 LUTs + 68 FFs
DRAM Access Control Violation	A malicious IP tries to illegally access the secure DRAM region	CWE-1257, CWE-1190, CVE-2022-37302	Confidentiality	Identify illegal access to the secure memory region by monitoring read/write requests	N/A	33 LUTs + 21 FFs
Unauthorized Cache Flush Instruction Execution	A guest OS illegally executes cache flush instruction from lower privilege levels	CWE-250, CWE-1281	Confidentiality	Observe the Control & Status registers to detect illegal instruction execution	MSTATUS register watch	8 LUTs + 6 FFs
Illegal Memory Page Access	A guest OS illegally tries to modify unauthorized memory pages	CWE-284, CWE-1262	Confidentiality	Observe the Control & Status registers to detect illegal memory page access	MSTATUS register watch	8 LUTs + 9 FFs
AES Denial-of-Service	A Trojan tracks incoming patterns (plaintext) and HALT will be occurred once a specific pattern is observed.	AES-T500, BASICRSA-T200, BASICRSA-T400	Availability	the valid_out signal must become high after n number of clock-cycle (once plaintext is refreshing).	AES FSM watch (counter-based)	9 LUTs + 10 FFs
AES Information Leakage	A Trojan leaks the AES secret key through the common bus in the SoC (Triggering with a specific plaintext)	AES-T1300, CVE-2018-8933, CVE-2014-0881	Confidentiality	No cipher key (partial or full) should be visible through the primary output	Cipher key and Ciphertext Observation	50 LUTs + 8 FFs

To ensure a robust security policy, **the flushing instruction should always execute from the highest privilege level (M mode)**. It is crucial to maintain a secure system and prevent any unauthorized access to sensitive information. Therefore, the *TVM* bit should be set to prevent any execution of *SFENCE.VMA* from lower privilege levels.

```
check_instruction_reg;
if program_counter_value_matches_SFENCE.VMA_instr
    check_TVM_bit_in_MSTATUS_reg;
    if TVM == 0
        interrupt = 1;
```

(Vulnerability 4) Illegal Memory Page Access: The issue of illegal memory page access is closely linked to vulnerability 3 and involves the *MSTATUS* register of the processor. Generally, user-level virtual memory pages are inaccessible from the supervisor privilege mode (S mode). The Supervisor Memory Access (*SUM*) bit in the *MSTATUS* register determines whether the supervisor mode can access user pages. When the *SUM* bit is set, any guest OS in supervisor mode can read, write, or modify the information in a user page from a different privilege level. This can create serious security issues when used in a multiple Host system where the data from different parties are secret from one another. On the other hand, if the *SUM* bit is cleared, any attempt to modify user pages from different privilege levels generates a data access error. To prevent unauthorized access, **it is crucial to clear the SUM bit if user page access is restricted**. This action ensures that only authorized access is granted, and any attempt to access the user pages from a higher privilege level is denied.

```
if !(S_mode_has_virtual_memory_access)
    check_SUM_bit_in_MSTATUS_reg;
    if SUM == 1
        interrupt = 1;
```

(Vulnerability 5) AES Denial of Service: This vulnerability is Trojan-enabled, in which the Trojan is triggered by a sequence of specific patterns in the plaintext. This is also an example of software-triggered vulnerability. If the plaintext matches the trigger patterns, the whole process goes into a denial of service. AES module cannot generate a *valid_out* signal while in a DoS situation (HALT in operation). In this case, the policy is that **the valid_out signal must become high after n number of clock-cycle**, where n is the number of clock-cycle needed to complete the encryption (high-level showing in the following snippet).

As AES is a security-critical IP [44], we used an SSM to monitor the AES internal signals. For this vulnerability, the SSM collects the AES *valid_out* signal data and sends it to the eFPGA. We implemented the security policy in eFPGA which detects this DoS vulnerability.

```
for all clock_cycles do
    if valid_input
        count = 0;
    else if !valid_out
        count = count+1;
    if count > n
        interrupt = 1;
```

(Vulnerability 6) AES Information Leakage: Similar to vulnerability 5, this vulnerability is Trojan-enabled, and when triggered, the cipher key or parts of the cipher key will be leaked through the primary outputs (within the ciphertext). In this specific Trojan, the trigger for this Trojan is the specific pattern in the plain text, and the payload is the cipher key leakage. The SSM dedicated to the AES module will check the AES data, including plaintext, cipher key, and the ciphertext for catching this vulnerability[7]. By sending the data collected and encoded in the SSM, the policy added into the eFPGA will detect whether the Trojan is triggered or not. In this case, the policy states that **no cipher key (or its parts) should be visible through the primary output** (shown in the following snippet).

```
byte_0_match = 0; j = 0;
for all ciphertext_bytes do
    if !byte_0_match && (ciphertext_byte[i] ==
         cipherkey_byte[0])
        byte_0_match = 1; j = 1;
    if byte_0_match == 1
        if ciphertext_byte[i+1] = cipherkey_byte[j]
            byte_match = 1; j = j+1;
        else byte_match = 0;
    if byte_match && (j == cipherkey_length-1)
        interrupt = 1
```

In all use case scenarios, the *EnSAFe* framework follows an identical procedure. Fig. 7 shows the overall steps of detecting the vulnerabilities/threats using an eFPGA and SSM. Per each use case scenario, the CPU has the right to send a trigger signal to the eFPGA to start monitoring. If the security policy requires the usage of an SSM (Vulnerability 3 to 6), a set of commands will be sent to the SSM from eFPGA that determines which event to be monitored. The SSM then gathers all the signals into the event tracker buffers and sends the encoded data back to the eFPGA. Finally, the eFPGA, with minimal logic resources, will complete the monitoring for detecting potential vulnerabilities. If the security policy does not involve an SSM for monitoring

[7]As the cipher key is a security-critical entity, comparing the output with each byte of the key is performed inside the SSM.

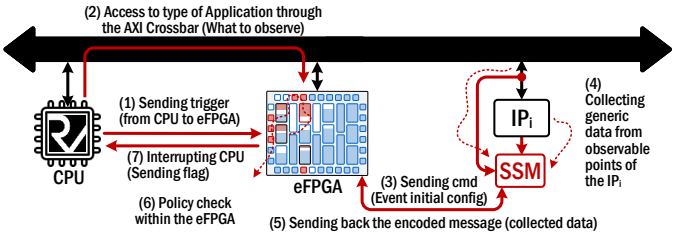


Fig. 7: Typical Steps of Vulnerability Detections in *EnSAFe*.

(Vulnerability 1 and 2), the policy in the eFPGA does the event monitoring, perform logic operations and decision making to detect the security vulnerability.

VI. Results and Discussions

As discussed (Table I), for larger eFPGAs, more unutilized resources are available. Table III shows what resources are required per each vulnerability to implement the policy into the eFPGA. As shown (the last column), only tens (10-70) LUTs/FFs are required per policy. Here, each policy is implemented into the eFPGA separately and thus the resource utilizations are calculated. However, we have optimized the implementations of the policies of rows 3 to 6 (Vulnerability 3 to 6) in Table III, and the results are shown in Table IV.

A. Implementation Results

Implementation Results of V1: Vulnerability 1 (Rowhammer) involves DRAM bus monitoring. As the eFPGA is also used for acceleration, it has access to DRAM for high-speed operations. Hence, no separate SSM is required for this policy. Thus, the policy implementation does not need further optimization and so the resource utilization of this policy remains the same (row 1 of Table III and Table IV). The security policy for Vulnerability 1 directly observes the DRAM bus, detecting excessive read/write accesses against a threshold (see Section V-B). Fig. 8 shows the policy's waveform, generated from Xilinx Vivado, including DDR3 bus signals and the interrupt output. The policy's counter logic tracks read/write accesses within a set time window. The counter value exceeding the threshold triggers a processor interrupt. The time window and threshold, specific to DDR3 technology, define the critical frequency of read/write accesses causing Rowhammer [43]. These parameters can be in-field updated via eFPGA reprogramming.

Currently, the security enforcement action is assumed to be taken care of by the processor. Thus, the security policy lets the processor know about this detected vulnerability through the generated interrupt. As seen from Fig. 8, the interrupt is generated one clock cycle after the counter time window completes (when $time\_window\_active$ goes $high$ to low in Fig. 8). As this Rowhammer detection scheme is entirely based on hardware and by directly monitoring the DDR3 bus,

TABLE IV: SSMs & Optimized Policy Implementations - Resource Utilization

Vulnerability	SSM Event	Resource Needed for the SSM (Cell Count)	FPGA-based Resource Needed for the Policy
V1	N/A	N/A	61 LUTs + 68 FFs
V2	N/A	N/A	33 LUTs + 21 FFs
V3 and V4	MSTATUS register watch	21,529	10 LUTs + 12 FFs
V5 and V6	AES FSM and I/O watch	37,647	19 LUTs + 34 FFs

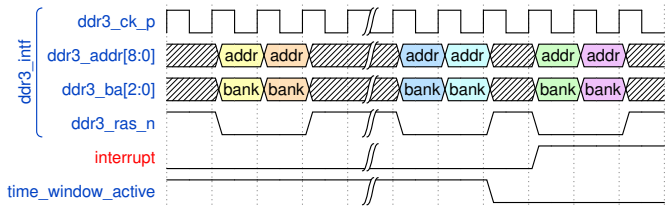


Fig. 8: Waveform of the signals for Rowhammer detection.

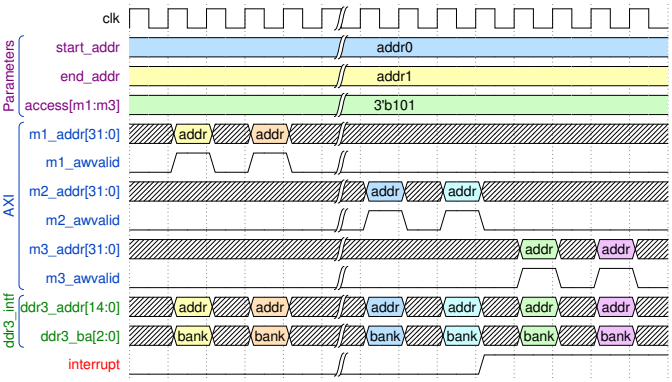


Fig. 9: Waveform of the signals for detecting DRAM access control violation.

this approach is reasonably faster than any software-based Rowhammer detection approach. The processor, upon reception of the interrupt, calls the ISR written for Vulnerability 1. We have developed the policy implementation and the ISR in such a way that the ISR can read the policy status register to know the memory address range that are being rowhammered. The ISR then starts DRAM refresh procedure for the corresponding memory bank as part of the security enforcement action.

Implementation Results of V2: Vulnerability 2 requires monitoring the DRAM bus as well as the system bus to detect DRAM access control violations and to identify access requesting sources. Following the same assumption as for the policy implementation of V1, the eFPGA will have access to the DRAM bus and the system bus as it will have an accelerator application. Hence, we do not need an SSM to collect data for this implementation. So, the policy implementation does not need further optimization as it can directly monitor the two interconnects. Thus, the resource utilization of this policy implementation remains the same (row 2 of Tables III and IV).

Fig. 9 displays the signal waveforms for our policy, which protects a defined memory region and regulates access for each bus master. We used an AXI4 system bus with 3 masters, and the parameters are in-field upgradable via eFPGA reprogramming. These parameters could be software-updated with minor policy modifications. The policy involves monitoring select AXI bus (address and valid signals) and DRAM bus (bank and memory address signals) signals, shown in "blue" and "green" respectively in Fig. 9. Access requests to the protected region are checked against the master's access policy. Unauthorized attempts trigger an interrupt, and the ISR identifies the violating master through the policy status register. The ISR then sends configuration bits to the AXI4 XBAR IP to temporarily disable the bus master port that sent the request. Finally, the ISR resets and reconfigures the bus master (i.e., DMA) by calling the corresponding IP default configuration and programs AXI4 XBAR IP again to enable the bus master port.

Implementation Results of V3 and V4: Vulnerabilities 3 and 4, related to RISC-V Ariane core processor, both require

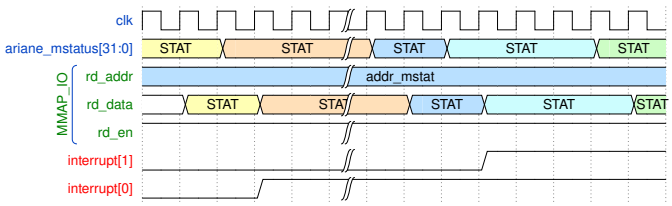


Fig. 10: Waveform of the signals for detecting Ariane core vulnerabilities.

MSTATUS register monitoring. These policies are merged into one implementation using a single SSM for the MSTATUS register, with resource utilization detailed in Table IV, row 3. We synthesized SSMs using the Skywater 130nm library for ASIC Place & Route flow. The SSM collects and stores 32-bit MSTATUS data in internal registers. The security policy uses a read-only memory-mapped I/O for SSM connection, as MSTATUS is read-only and requires no configuration bits. Fig. 10 shows the signals and their waveforms for this setup.

The security policy, synthesized and loaded onto the eFPGA, monitors data from the SSM to detect illegal cache flush instructions and unauthorized memory page access. It checks the TVM bit (bit 20) and SUM bit (bit 18) in the 32-bit SSM data for these vulnerabilities. Detected vulnerabilities trigger separate interrupts for each, with $interrupt[0]$ for V3 and $interrupt[1]$ for V4, as illustrated in Fig. 10. In case of an interrupt generated for V3, the ISR checks the privilege level. If the privelege level is not the highest privilege level, the ISR writes 1 to the TVM bit of the MSTATUS register. Similarly, in case of an interrupt generated for V4, the ISR checks the privilege level. If the privilege level is supervisor, then the ISR reads the SUM bit of the MSTATUS register. If the bit is 1, the ISR clears the bit by writing 0.

Implementation Results of V5 and V6: Vulnerabilities 5 and 6 relate to AES IP, with Vulnerability 5 involving $valid\_in$, $valid\_out$, and AES-internal register monitoring, while Vulnerability 6 focuses on cipher key and ciphertext I/O. A combined policy implementation for these vulnerabilities uses one SSM to monitor AES I/O and internal registers, with resource utilization detailed in Table IV, row 4. Fig. 11 illustrates the policy implementation waveforms, showing the SSM's connection to AES IP and monitoring of $valid\_in$, $valid\_out$, and $data$ signals (displayed in Fig. 11). The eFPGA security policy interacts with the SSM via memory-mapped I/O (MMAP_IO) with separate read/write channels, indicated in "green" in Fig. 11. The write channel configures the SSM, including event detection logic activation. Once enabled, the SSM collects data, read by the eFPGA, to detect these vulnerabilities. Vulnerability 5, an AES denial of service vulnerability, is identified by measuring the delay of the $valid\_out$ signal after $valid\_in$ assertion against the number of clock cycles required for encryption. An excessive delay of the $valid\_out$ triggers an $interrupt[0]$ output (seen in Fig. 11). The corresponding ISR calls the AES reset function to perform software reset and to set default configuration of the IP. Subsequently, the ISR calls the AES default test function to verify the successful execution. Upon passing the test, the ISR logs the event status and proceeds to return. The execution of the user application is then restarted. Vulnerability 6 denotes the AES information leakage vulnerability discussed in Section V-B. To detect this vulnerability, the security policy checks each byte of the ciphertext output of the AES and compares it with the cipher key. Similar to previous policy implementations, upon detection

TABLE V: Resource Utilization (post-Synthesis) for Implementing Different Use Cases (Acceleration vs. Security) on the eFPGA Fabric

Resources	Acceleration Use Cases					Security Policies			
	Custom I/O	Wide I/O	BNN	CRC	Elliptic	V1	V2	V3+V4	V5+V6
GPIO	72	192	384	64	192	———	———	44*1	———
LUT	355	565	1465	122	4064	61	33	10	19
FF	240	550	935	41	2122	68	21	12	34
eFPGA size	———	32×32*2	———		28×28	———		5×5*3	———

*1: 44 is the total I/O needed for all Vulnerabilities (together). V_i: Vulnerability i
*2: 32×32 is the eFPGA fabric for all I/Os, BNN, and CRC.
*3: 5×5 is the eFPGA fabric for all security policies together.

of this vulnerability, the security policy generates $interrupt[1]$ (shown in Fig. 11). The ISR corresponding to this interrupt bit stops execution of the user application, resets the AES IP, and then proceeds to generate a new key for AES encryption.

B. Resource Utilization Analysis

Our security policies (which behave like synthesizable assertions) can easily fit into the unutilized resources. With this low overhead, we can develop a more comprehensive security policy engine that monitors a wide range of policies within the eFPGA. Table V provides a comparative illustration between the resources required for modules mapped to the eFPGA in Arnold SoC vs. that of security policies. The BNN accelerator, a simple version of the accelerator in [45], is used as an acceleration use case in Table V. It has four interconnects to the main memory to optimize bandwidth. The accelerator assumes a 3-D array of integer structures for the input layers and filters (number of filters × rows × columns). Although it is a simple implementation, it still requires a large number of eFPGA resources. The acceleration use cases in Table V require resources over six times greater than those for security. This results in a large, underutilized eFPGA fabric for accelerators, enabling its repurposing for security without overhead. Table V shows that accelerator applications (individually) need much larger fabrics than security policies. For example, Arnold requires a 32×32 fabric (with over 6K LUTs, 4K FFs, and DSPs), whereas security policies in Table III need just a 5×5 fabric, indicating a 40:1 ratio. Future expansions to the list of use cases will show this ratio remains largely unaffected even with additional security policies.

Table VI provides a breakdown of area overhead for various RISC-V Ariane SoC modules, using separate eFPGA fabrics for security and acceleration applications, normalized to the CPU size. The eFPGA for acceleration is about 46x larger than the CPU, aligning with Arnold SoC's findings [1] where eFPGA is ∼80% of die size. In contrast, the eFPGA for security is roughly 2.5x smaller than the CPU, indicating that

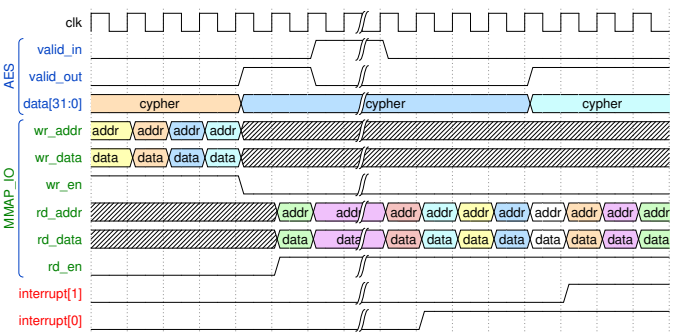


Fig. 11: Waveform of the signals for detecting AES-internal vulnerabilities.

TABLE VI: Area Distribution of the Some Main Components of the SoC

Module	Area [μm^2]	Cell Count	Normalized Ratio (w.r.t CPU)
CVA6 (CPU)	1,734,757	216,922	1
eFPGA for Sec. Policy	695,638	79,365	0.401
eFPGA for Acceleration	79,715,267	11,085,611	46.004
AES Core	902,325	110,375	0.520
SSM (x2)	523,115	59,176	0.300
Direct Memory Access	167,780	13,572	0.096
SPI Peripherial	12,827	1,269	0.007

the low-utilization eFPGA fabric can be repurposed for non-intensive tasks like security monitoring with minimal overhead. Moreover, the SSM's area overhead in Table VI reveals that adding monitoring modules per IP incurs negligible area. The overhead primarily depends on the event buffer size, which, for targeted vulnerabilities, is optimized to maintain low overhead.

C. Vulnerability Detection Timing

Fig. 12 shows the detection times for vulnerabilities in *EnSAFe*, measured in clock cycles. During simulations, vulnerabilities (V2∼V6, as detailed in Section V-B) were randomly triggered. Each was detected within 10 clock cycles, with detection times consistent regardless of when vulnerabilities triggered. The V1 line graph, indicating Rowhammer detection time, measures the cycles from the end of the time window to the interrupt generation. The V1_1 line graph represents the count of clock cycles after the read/write accesses surpass a certain threshold. Since these read/write actions are introduced randomly in time to different memory banks, the V1_1 line graph fluctuates between 34 and 107. This variability arises as we await the end of the time window before comparing it with the corresponding threshold value and the interrupt generation.

D. Policy Replacement at Runtime

We showcased six vulnerability scenarios, but there may be undiscovered ones exploitable in the field. The eFPGA's reconfigurability within the extended *EnSAFe* framework allows on-the-fly security policy updates. For instance, the 32-bit MSTATUS register, where each bit indicates an illegal state and a potential vulnerability [33], includes the *Modify PRiVilege* (MPRV) bit. This bit dictates the privilege level for memory operations: MPRV = 0 maintains the current level, while MPRV = 1 elevates it for load/store instructions. A Hardware Trojan altering MPRV during a lower privilege operation could compromise security-critical data. We counter this by adding a policy to prevent MPRV modification during debugging. After updating the *EnSAFe* framework and eFPGA bitstream, the revised bitstream is sent via AXI interconnect to the CPU. The FPGA Control Unit's half-DMA module adjusts the eFPGA core logic to the new policy, enhancing protection against zero-day and evolving threats.

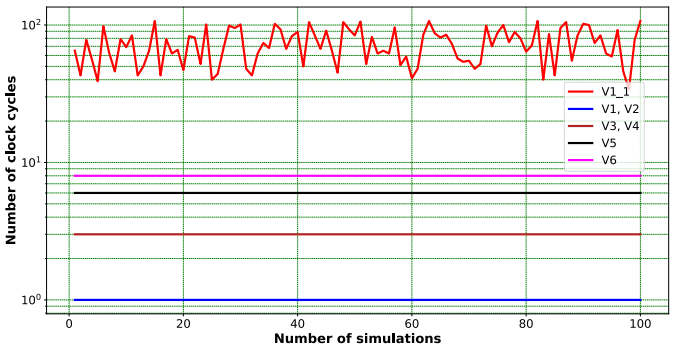


Fig. 12: Vulnerability detection time of the policy implementations for 100 random simulations.

```
if !(Privilege_level_in_M_mode)
    check_MPRV_bit_in_MSTATUS_reg;
    if MPRV == 0
        interrupt = 1;
```

In this experiment, we implemented security monitoring for the six vulnerabilities outlined in Table III. However, when dealing with certain accelerator implementations, there might be insufficient unutilized resources in the eFPGA to accommodate all six security policies. In such instances, we may need to consider implementing a reduced number of policies while ensuring the same level of observability [46] for the eFPGA. For our specific use cases, it's noteworthy that security policies for Vulnerabilities 1 and 2 do not necessitate the use of an SSM. Assuming that the eFPGA has access to the DRAM bus and the system bus for accelerator implementation, we can seamlessly deploy these policies at run-time without modifying the I/O connections of the eFPGA. This adaptability becomes crucial in scenarios where Vulnerability 1 or 2 emerges as an unforeseen threat that the designer had not initially considered. Furthermore, if the eFPGA possesses the required hardware observability [46], either directly or through the SSM, and there are sufficient unutilized resources available in the eFPGA for the revised security policy implementation, addressing unforeseen vulnerabilities at run-time becomes feasible, even when the chip is already deployed in the field. This emphasizes the importance of the system's ability to dynamically reprogram the eFPGA, offering flexibility and responsiveness in managing unexpected security issues.

E. Comparison with Existing Works

Finally, we compare our proposed monitoring technique with existing solutions. As the existing solutions did not implement their techniques on benchmark SoCs and the targeted vulnerabilities are different in most existing solutions, a PPA (Power, Performance, and Area) based analysis and comparison among the solutions is not feasible. Hence, we adopt the quality attributes proposed in [46] for an ideal security monitoring solution, namely *scalability*, *sustainability*, *adaptability*, *observability*, and *distributability*, and compare our technique with existing solutions based on these quality attributes. For instance, our method is capable of identifying various vulnerabilities, unlike [14]–[16], which is limited to specific ones due to scalability issues. Unlike [10], [12], [13] that lack runtime upgradability, *EnSAFe* offers greater sustainability. Furthermore, the positioning of the SSMs in our method facilitates more effective monitoring of security-critical signals across different IPs, a capability that other methods [15], [16] do not offer. Table VII shows the comparison results where the last row corresponds to extended *EnSAFe* proposed here.

F. Limitations and Future Work

The effectiveness of the proposed extension of *EnSAFe* depends on the eFPGA's ability to observe signals crucial for policy implementation. During the design phase, a specific set of signals, identified by the threat model, is selected for monitoring by the eFPGA, either directly or through the SSM, to

TABLE VII: Comparison among Existing Solutions and Proposed Security Monitoring Solution to Meet the Quality Attributes

Security Monitoring Solutions	Quality Attributes				
	Scalability	Sustainability	Adaptability	Observability	Distributability
Security architecture for embedded systems (SAFES) [29]	LOW to MEDIUM	HIGH	LOW to MEDIUM	MEDIUM	MEDIUM
Infrastructure IP for secure SoC (IIPS) [6]	HIGH	LOW	MEDIUM	HIGH	LOW
Flexible architecture for security policy implementation [7]	LOW to MEDIUM	MEDIUM	MEDIUM to HIGH	MEDIUM	LOW
Exploiting design-for-debug for SoC security [8]	HIGH	LOW	MEDIUM	HIGH	LOW
SoC security architecture and CAD framework [9]	HIGH	HIGH	MEDIUM to HIGH	MEDIUM	LOW
Policy enforcement for secure IoT [10]	LOW to MEDIUM	LOW	MEDIUM to HIGH	HIGH	HIGH
Energy-efficient hardware monitors for secure information flow [11]	MEDIUM	N/A	MEDIUM	MEDIUM to HIGH	HIGH
Embedded system security through hardware isolation [12]	MEDIUM	LOW	LOW to MEDIUM	HIGH	MEDIUM
System-on-Chip security assertions [13]	HIGH	N/A	LOW	HIGH	HIGH
UltraSoC bus monitoring solution [14]	LOW to MEDIUM	LOW	MEDIUM to HIGH	MEDIUM to HIGH	HIGH
Programmable hardware monitor (PHMon) [15]	LOW	LOW to MEDIUM	LOW	LOW to MEDIUM	LOW
Programmable monitor for security enforcement (ProMiSE) [16]	LOW	LOW to MEDIUM	LOW to MEDIUM	LOW to MEDIUM	LOW
Extended *EnSAFe*	MEDIUM to HIGH	MEDIUM to HIGH	LOW to MEDIUM	HIGH	MEDIUM to HIGH

enforce the specified policies. However, attempting to establish connections between all security-critical signals (which can be a lot depending on the threat model) of the SoC and the eFPGA or the SSM is both impractical and unfeasible. Consequently, there exists a potential scenario where a runtime vulnerability requires monitoring a signal inaccessible to the eFPGA. In such cases, implementing a policy to address that particular vulnerability becomes unattainable. This limitation is intrinsic to the proposed technique but can be mitigated by adopting a thoughtful approach in selecting a suitable threat model. The key lies in meticulously choosing signals for monitoring based on the identified threat model, ensuring a more practical and effective implementation of security policies.

When a vulnerability is detected, an interrupt is generated for the processor. In our implementation use cases, the processor, upon reception of the interrupt, calls the ISR code written for that vulnerability. The ISR then performs the security enforcement action. There is a delay of several clock cycles between interrupt generation and the initiation of security enforcement, as the processor must process the interrupt and execute the corresponding ISR code. Thus, security enforcement is not instantaneous, allowing some time for illegal application/attack to progress. However, it's important to note that this delay is relatively small, and in our implementations, we observed it to be in the range of tens of clock cycles for V2∼V6 and within 100∼200 clock cycles for V1. To eliminate this delay and achieve an immediate or preemptive security enforcement action, implementing a preemptive vulnerability detection mechanism employing machine learning (ML) algorithms would be necessary. It's crucial to acknowledge that our current security monitoring technique does not involve any ML-based analysis in the eFPGA, constituting a limitation of the proposed approach. Our future efforts will focus on incorporating ML capabilities into our security monitoring framework to address this limitation and provide a more proactive security response.

VII. Conclusion

Relying on the fact that large-size eFPGA fabrics used for acceleration/efficiency in modern SoCs suffer from low utilization, in this paper, we extended and improved the *EnSAFe* framework for security monitoring of the SoCs' vulnerabilities with upgradability over time, which is the most suited solution for zero-day attacks. With the optimized insertion of SSMs, *EnSAFe* provides a distributed monitoring capability with observability into security-critical IPs and interconnects while the core security policies are mapped into the eFPGA fabric. To evaluate the efficiency of the extended framework, we showcased six different use cases of *EnSAFe*. Based on a comprehensive hardware threat analysis, we first developed a security policy database for the SoC. Following this, we methodically demonstrated the complete workflow by implementing, optimizing, and monitoring the security policies using *EnSAFe*. As *EnSAFe* benefits from the unutilized resources of already-integrated eFPGA, it enables the designers to support both acceleration and security monitoring on the same fabric with a low overhead.

References

[1] P. S. *et al.*, "Arnold: An eFPGA-augmented RISC-V SoC for flexible and low-power IoT end nodes."
[2] F. Renzini *et al.*, "A fully programmable eFPGA-augmented SoC for smart power applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 2, pp. 489–501, 2019.
[3] P. Mohan *et al.*, "Hardware Redaction via Designer-Directed Fine-grained eFPGA Insertion," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2021, pp. 1186–1191.
[4] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE S&P*, 2019, pp. 1–19.
[5] K. Z. Azar *et al.*, "Fuzz, Penetration, and AI Testing for SoC Security Verification: Challenges and Solutions," *Cryptology ePrint Archive*, 2022.
[6] X. Wang *et al.*, "IIPS: Infrastructure IP for secure SoC design," *IEEE Transactions on Computers*, vol. 64, no. 8, pp. 2226–2238, 2014.
[7] A. Basak *et al.*, "A flexible architecture for systematic implementation of SoC security policies," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 536–543.
[8] A. Basak *et al.*, "Exploiting design-for-debug for flexible SoC security architecture," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.
[9] A. Nath *et al.*, "System-on-chip security architecture and CAD framework for hardware patch," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 733–738.
[10] F. Siddiqui *et al.*, "Embedded policing and policy enforcement approach for future secure iot technologies," in *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, 2018, pp. 1–10.
[11] S. Sefton, T. Siddiqui, N. S. Amour, G. Stewart, and A. K. Kodi, "Garuda: Designing energy-efficient hardware monitors from high-level policies for secure information flow," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2509–2518, 2018.
[12] S. Kumar Saha and C. Bobda, "Fpga accelerated embedded system security through hardware isolation," in *2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2020, pp. 1–6.
[13] Y. Lyu *et al.*, "System-on-chip security assertions," *arXiv preprint arXiv:2001.06719*, 2020.
[14] "UltraSoC Embedded Analytics for SoCs," https://www.linkedin.com/company/ultrasoc-technologies/?originalSubdomain=uk, accessed: Aug 01, 2023.
[15] L Delshadtehrani *et al.*, "PHMon: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 807–824.

[16] X. Wang, L. Feng, and Z. Wang, "Promise: A high performance programmable hardware monitor for high security enforcement of software execution," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2023.

[17] S. Mohammad *et al.*, "Required policies and properties of the security engine of an soc," in *2021 IEEE International Symposium on Smart Electronic Systems (iSES)*, 2021, pp. 414–420.

[18] N. Farzana *et al.*, "SoC Security Properties and Rules," *Cryptology ePrint Archive*, 2021.

[19] M.M.M. Rahman *et al.*, "Ensafe: Enabling sustainable soc security auditing using efpga-based accelerators," in *36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2023, pp. 1–6.

[20] X. Tang *et al.*, "OpenFPGA: An open-source framework for agile prototyping customizable FPGAs," *IEEE Micro*, vol. 40, no. 4, pp. 41–48, 2020.

[21] D. Halder, M. Merugu, and S. Ray, "Obnocs: Protecting network-on-chip fabrics against reverse-engineering attacks," *arXiv preprint arXiv:2307.05815*, 2023.

[22] K. Z. Azar *et al.*, "From cryptography to logic locking: A survey on the architecture evolution of secure scan chains," *IEEE Access*, vol. 9, pp. 73 133–73 151, 2021.

[23] H. M. Kamali *et al.*, "Advances in Logic Locking: Past, Present, and Prospects," *Cryptology ePrint Archive*, 2022.

[24] N. N. Anandakumar *et al.*, "Rethinking watermark: Providing proof of IP ownership in modern socs," *IACR Cryptol. ePrint Arch.*, p. 92, 2022.

[25] M. M. M. Rahman *et al.*, "Capec: A cellular automata guided fsm-based ip authentication scheme," in *2023 IEEE 41st VLSI Test Symposium (VTS)*, 2023, pp. 1–8.

[26] J. Rajendran *et al.*, "Security analysis of integrated circuit camouflaging," in *ACM SIGSAC conference on Computer & communications security*, 2013, pp. 709–720.

[27] L. Masure, C. Dumas, and E. Prouff, "A comprehensive study of deep learning for side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 348–375, 2020.

[28] H. Al-Shaikh *et al.*, "Sharpen: Soc security verification by hardware penetration test," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 579–584. [Online]. Available: https://doi.org/10.1145/3566097.3567918

[29] G. Gogniat *et al.*, "Reconfigurable hardware for high-security/ high-performance embedded systems: The safes perspective," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 2, pp. 144–155, 2008.

[30] A. Sehgal, S. Goel, E. Marinissen, and K. Chakrabarty, "Ieee p1500-compliant test wrapper design for hierarchical cores," in *2004 International Conference on Test*, 2004, pp. 1203–1212.

[31] W. Stallings *et al.*, *Computer Security: Principles and Practice*. Pearson Upper Saddle River, 2012, vol. 2.

[32] H. Kyung *et al.*, "Performance monitor unit design for an axi-based multi-core soc platform," in *ACM Symposium on Applied Computing*, 2007, pp. 1565–1572.

[33] S. Tarek *et al.*, "Benchmarking of soc-level hardware vulnerabilities: A complete walkthrough," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2023.

[34] S.R. Rajendran *et al.*, "Hunter: Hardware underneath trigger for exploiting soc-level vulnerabilities," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[35] S. R. Rajendran *et al.*, "Exploring the abyss? unveiling systems-on-chip hardware vulnerabilities beneath software," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2024.

[36] MITRE, "HW CWEs," https://cwe.mitre.org/data/definitions/1194.html.

[37] "HACK@DAC'23," https://hackatevent.org/hackdac23/.

[38] N. Farzana, A. Ayalasomayajula, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Saif: Automated asset identification for security verification at the register transfer level," in *2021 IEEE 39th VLSI Test Symposium (VTS)*, 2021, pp. 1–7.

[39] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460.

[40] F. Zaruba *et al.*, "The cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[41] Skywater Technology, "SkyWater Open Source PDK," https://github.com/google/skywater-pdk, 2020.

[42] Digilent Genesys 2, "Genesys 2 FGPGA Development Board," https://digilent.com/reference/programmable-logic/genesys-2/start, 2016.

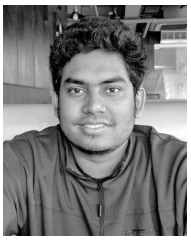
[43] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2020.

[44] T. Rahman *et al.*, "Design and security-mitigation of custom and configurable hardware cryptosystems," in *2023 IEEE 16th Dallas Circuits and Systems Conference (DCAS)*, 2023, pp. 1–6.

[45] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2020.

[46] M. M. M. Rahman *et al.*, "Efficient soc security monitoring: Quality attributes and potential solutions," *IEEE Design & Test*, pp. 1–1, 2023.

Mridha Md Mashahedur Rahman is a PhD student at the Electrical and Computer Engineering Department of the University of Florida. He received his B.Sc. degree in Electrical and Electronics Engineering from Bangladesh University of Engineering and Technology (BUET) in 2017. His research interests include hardware security and trust, security policy, and security monitoring.



Shams Tarek is currently a Ph.D. student under the supervision of Dr. Farimah Farahmandi at the Electrical and Computer Engineering Department at the University of Florida since 2021. He received his B.Sc. degree in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology (BUET) in 2019. His research focuses on hardware security and trust.

Kimia Zamiri Azar is a research assistant professor in the Department of Electrical and Computer Engineering at the University of Florida. She received a Ph.D. degree from the Department of ECE at George Mason University in 2021. She also received her M.S. and B.S. from the Department of ECE at Shahid Beheshti University, 2015, and K. N. T. University, 2013, respectively. Her research interests span hardware security and trust, supply chain security, System-on-Chips security validation and verification, and IoT security.

Mark Tehranipoor is currently the Intel Charles E. Young Preeminence Endowed Chair Professor in Cybersecurity and the Chair of the Department of Electrical and Computer Engineering (ECE) at the University of Florida. His current research projects include: hardware security and trust, supply chain security, IoT security, VLSI design, test and reliability. He is a recipient of a dozen best paper awards and nominations. He served as the founding Director for Florida Institute for Cybersecurity (FICS) Research from 2015-2022.

Farimah Farahmandi is an assistant professor in the Department of ECE at the University of Florida. She received her Ph.D. from the Department of CISE at the University of Florida, 2018. She received her B.S. and M.S. from the Department of ECE at the University of Tehran, Iran in 2010 and 2013, respectively. Her research interests include design automation of System-on-Chips and energy-efficient systems, formal verification, hardware security validation, and post-silicon validation and debug.