Cultivating Security: Debug Authentication for Ensuring the Security of SoC's Root of Trust

Arash Vafaei, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi
Dept of Electrical and Computer Engineering, University of Florida
Gainesville, Florida
Email: {arash.vafaei, sujansaha}@ufl.edu, and {tehranipoor, farimah}@ece.ufl.edu

***Abstract*—Hardware-assisted debugging provides the necessary infrastructure for developers to closely monitor program behaviors at the microarchitectural level in a system-on-chip (SoC). However, debug infrastructure jeopardizes the security of the system by providing a backdoor for accessing crucial assets embedded in the system because of the inevitable increase in observability. While trusted execution environments (TEE) provide an extra level of security and isolate design assets, the security implication of hardware debug integration on TEEs has not been investigated. In this paper, we introduce a multi-level bidirectional access authentication mechanism over the debug module that defines the minimum number of privilege levels needed and the access details at each level so that debug users are authorized and blocked from accessing assets private to other entities. Trust is established by exchanging certificates both from the debugger and SoC sides to implement a bidirectional authorization platform in order to restrict the debugger's access to SoC assets as well as prevent the debugger's test data from being accessed by an SoC impersonator through emulation. We provide a prototype of the debug authentication platform on RISC-V architecture that proves the small overhead of the approach while staying compatible with traditional debug efforts.**

I. Introduction

With the increasing complexity of modern System-on-Chip (SoC) designs—featuring multi-core processors, deep memory hierarchies, numerous on-chip peripherals, and Trusted Execution Environments (TEEs)—debugging these systems has become more difficult. Hardware-assisted debugging is a vital tool that allows engineers to observe internal behaviors in real time without modifying software execution, making it more effective than traditional software debugging, which often distorts system behavior by inserting payloads. However, while hardware debugging improves visibility into the system, it also introduces new attack surfaces that can compromise the confidentiality and integrity of sensitive data [14, 8].

Hardware debug interfaces such as JTAG and scan chains allow direct access to internal signals of IP blocks, enable arbitrary instruction execution, and permit full memory extraction [18]. These powerful features can be misused by attackers to leak encryption keys, tamper with memory contents, or even inject malicious payloads. For instance, AES and RSA modules have been shown to be vulnerable to scan-chain-based attacks that expose round keys during operation [1, 19]. Debug interfaces also allow attackers to pause execution and inject instructions that can result in privilege escalation or unauthorized memory access [11]. Firmware extraction is another threat where adversaries can read or patch firmware through debug ports for reverse engineering or bypassing security checks [18, 16].

These risks are especially concerning in manufacturing and field-deployed environments, where sensitive assets such as encryption keys, secure firmware, and test seeds may be exposed over debugging paths. To mitigate such threats, various countermeasures have been proposed. One set of approaches involves encrypting the debug data path using stream or block ciphers to protect information in transit [17, 16]. Others enforce authentication using passwords or challenge-response mechanisms. Password-based debug control, while simple, lacks strong identity binding [10], whereas PUF-based authentication methods offer stronger guarantees by binding debugger access to unique hardware fingerprints [7].

Another strategy is to permanently disable debug interfaces after manufacturing using e-fuses [11]. While this prevents unauthorized access, it also eliminates in-field debugging capabilities. Overall, most solutions fall into three categories: encrypting data, authenticating debuggers, and enforcing access control through monitoring [12]. These methods often come with trade-offs in area, performance, or flexibility, and many remain vulnerable to reverse engineering.

To address these challenges, this paper proposes a certificate-based secure debugging framework for SoCs that enables controlled and authenticated access across the product lifecycle. As illustrated in Figure 1, different stakeholders—including test engineers, field service providers, and security analysts—require varying levels of debug access. Our proposed solution enables mutual authentication between the SoC and debugger using certificates and provides role-based privilege levels to enforce access policies dynamically.

The contributions of this work are as follows:

- Introducing a mutual authentication protocol based on certificates to ensure only authorized debuggers can interact with SoC debug infrastructure.

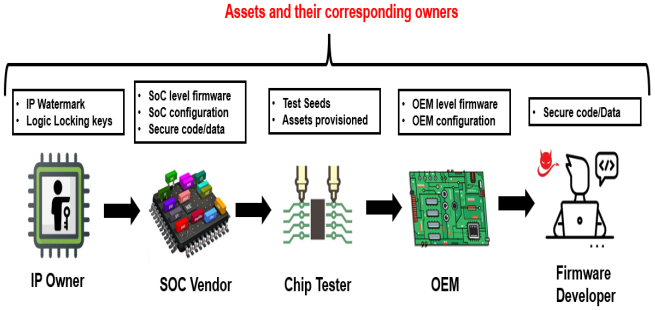

Fig. 1. Various entities using debug infrastructure and the attack targets (assets) at each stage.

- Defining multiple privilege levels to allow differentiated access for users based on operational domain (e.g., secure vs. non-secure world).
- Enforcing access control policies through runtime monitors and evaluating the proposed framework on a RISC-V SoC, including hardware overhead analysis.

The rest of this article is organized as follows. Section II explains the concepts required to understand the solution, while Section III explains the access levels in general secure SoC architectures. The last Section will introduce the results for a RISC-V example.

II. Background and Preliminary Concepts

In this section, the main concepts that provide the essence for understanding the architectures, procedures, and authentication protocols involved in the authorization of an external debugger are explained in detail.

A. Threat Model

To understand debug-related threats, it's crucial to identify the various users of the debug interface throughout the SoC lifecycle. Entities such as testers, RMA operators, and firmware developers require different levels of debug access. However, this access can be misused to bypass physical memory protection and compromise confidential assets, such as Boot ROM code or OEM-proprietary data. Non-secure developers must never access secure programs or data, making strict isolation between secure and non-secure worlds essential. Any debug access violating these isolation boundaries is a critical security vulnerability. Additionally, with the rise of remote debugging, the debugger itself can become a threat—transmitting private code and configurations to potentially unauthorized SoCs. Our threat model therefore addresses vulnerabilities from both SoC and debugger perspectives to ensure robust and granular access control across trusted and untrusted domains. Figure 1 illustrates stakeholders, assets, and their provisioning timeline throughout the SoC lifecycle.

B. Certificate-Based Authentication

Certificates, based on Public Key Infrastructure (PKI), are widely used for authentication due to their flexibility in dynamic validation and resistance to reverse engineering—advantages over hardware-based methods like PUFs. A certificate contains the public key of the issuing entity along with other identifying information, enabling secure communication. To prevent impersonation, certificates are digitally signed using the private key of a trusted authority. This signature is generated by first hashing the certificate content and then encrypting the hash using the authority's private signing key. This ensures the certificate's integrity—any modification changes the hash and invalidates the signature.

As an example, X509 certificate includes the issuer ID, public key, digital signature, and custom fields such as SoC ID and debugger privilege levels used in our secure debug framework. During validation, the system recomputes the hash from the certificate content (excluding the signature), then decrypts the provided signature using the corresponding public validation key. If the decrypted hash matches the computed

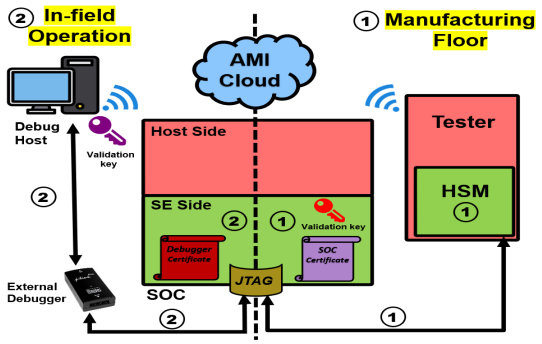

Fig. 2. The communications and assets are marked with number 1 for the manufacturing floor and number 2 for in-field operations. Each certificate and its validation key have the same color and AMI has a table that saves all the keys for both signing and validating as well as the whole certificate issued.

one, the certificate is deemed authentic and unaltered. This mechanism ensures trusted authentication of both debugger and SoC in the proposed secure debug infrastructure. The following section explains how keys and certificates are provisioned during manufacturing and deployment.

C. Secure SoC Architecture

To counter growing cyber threats, modern SoCs have adopted process isolation since 2018, confining attacks to vulnerable components and protecting critical assets. Central to this effort is the Trusted Execution Environment (TEE) or Security Engine (SE), an isolated hardware-supported environment that ensures confidentiality, integrity, and authenticity of sensitive data—even if the OS or hypervisor is compromised [6]. SEs either retain assets within their boundary or securely provide services to the host, such as encrypted data transfers [13]. TEE implementations include Intel SGX with enclave-based isolation and attestation [5], ARM TrustZone's secure/normal world partitioning [2], and Keystone for RISC-V, which supports memory encryption and remote attestation [9]. Recent extensions also protect against supply chain threats like IP theft through features such as watermarking [3], logic locking, and secure boot [13]. However, the integration of SEs introduces new vulnerabilities via the debug port, which often connects both secure and non-secure domains. This shared access can expose high-risk services to attackers. Existing defenses—such as disabling the debug interface or applying weak authentication—are either impractical or poorly managed by OEMs [11]. This paper investigates secure debugging in SE-equipped SoCs, focusing on protecting sensitive assets from unauthorized debug access.

D. Asset provisioning and Security Engine

The certificate signing and validation process involves two key phases. In the manufacturing phase, SoC-specific assets are provisioned, including the debugger certificate validation key and the SoC certificate. Since the tester is untrusted, a Hardware Secure Module (HSM) facilitates secure provisioning and establishes trust between the Asset Management Infrastructure (AMI) and the tester. The AMI, acting as the Certificate Authority (CA), generates key pairs and certificates, maintains a database indexed by SoC IDs, and issues certificates tailored to each chip [13]. Each SoC securely stores its

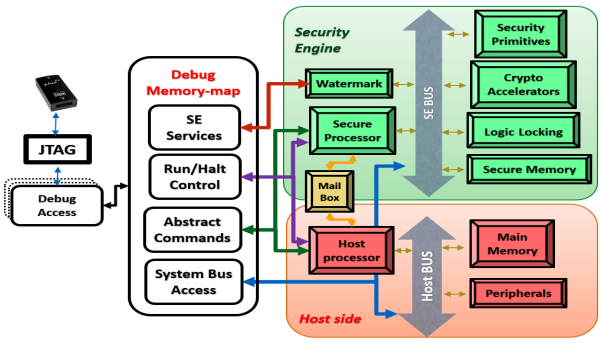

Fig. 3. Debug platform services and connections in the presence of a Security Engine in the SoC.

validation key and SoC certificate in a tamper-proof memory. In the in-field debug phase, the external debug host contacts the AMI and requests a certificate for a specific SoC ID, authenticating itself through standard internet protocols. The AMI then signs a debugger certificate with the private key linked to that SoC ID, embedding necessary debug parameters such as privilege level and a public encryption key for secure communication. The Security Engine (SE) inside the SoC uses the pre-provisioned validation key to verify the debugger certificate's authenticity. Figure 2 shows both phases, highlighting the secure certificate flow and validation path for authenticated debug access.

III. Secure Debug

With the foundational concepts of secure debug architecture and certificate validation established, we now examine common debug architectures to define a privilege-based access model that is independent of the underlying instruction set architecture (ISA). Understanding how a Security Engine (SE) integrates with the host-side SoC and shares debug infrastructure enables us to enforce isolation with minimal architectural changes.

In a secure SoC, the SE provides critical services such as cryptographic operations, watermarking, IP unlocking (for both SE and host-side IPs), and security primitives like True Random Number Generators (TRNG) and odometers [13]. The SE includes a secure processor hardened against attacks like cache timing and power side-channels. Certain services must be accessed via debug to maintain trust, especially when the host processor is untrusted. For instance, watermark extraction cannot safely be issued from an untrusted host processor. Furthermore, secure firmware development necessitates debug access to the SE processor for functionality verification and bug tracing. On the host side, multiple cores typically require hardware-assisted debugging. Host-side buses and memory hierarchies are also exposed via debug, but SE-provided services such as memory encryption and protection units ensure the security of sensitive assets.

To generalize this architecture, Figure 3 presents a high-level view of a secure SoC. Debug services are accessed through a memory-mapped interface and routed to modules within the SE via specific debug port registers. Connections to processors and buses—illustrated using color-coded pathways—highlight the scope of debug interactions.

To secure debug access, a multi-level privilege model is introduced. This model governs read/write operations on debug services and restricts access based on entity roles within the supply chain. The minimum required privilege levels are derived from the threat model and connection paths shown in Figure 3. With the big picture of the SoC in mind, we can now define a multi-level access paradigm and how and which entities can utilize each of these levels for their debug purposes. These privilege levels differ by blocking specific accesses to certain entities on the supply chain and our discussion will be based on the connections presented in various colors in Figure 3.

The debug memory-map services have connections to different entities inside SoC architecture and the minimum number of privilege levels necessary to address the threat model mentioned so far consists of:

Level 0: Level zero is a special privilege access dedicated to IP owners. In case the IP owner wants to extract its IP watermark in a court of law or has provided a logically locked IP and demands to check the required key, or any IP-specific service that SE provides, this level guarantees secure access without host-side involvement through debug infrastructure.

Level 1: This level has full permission to access the connections on the SE side. It is considered a high-privilege level of access used by secure firmware developers, SoC owners, and any entity with private assets inside SE boundaries. When AMI grants this level of access to a debugger they can run code on the secure processor and access its registers and allocated memory. The only restriction at this level is special accesses reserved for IP owners such as watermark extraction services.

Level 2: In this level all the services including run/halt control, abstract command, and SE bus access are disabled while the mailbox configuration still allows for the host side code to issue requests to SE through secure channels already implemented inside mailbox IP. This will make debugging host-side code with calls to the security function possible while preventing access to assets inside SE by debug channels. The SE services shown with the red arrow are also disabled for this level.

Level 3: The only difference between this level and with previous level is the complete isolation of SE via disabling the mailbox. This level exists so that the AMI can disable access to SE in case a vulnerability exists that can be exploited by offloading tasks through the mailbox to SE. General developers who should be prohibited from accessing assets within SE boundaries will be issued a certificate with this privilege level of access.

Level 4: These levels are defined per host-side requirements and are based on the host SoC architecture. These levels are similar to Level 3 because they are also blocked from accessing assets within the SE boundary but are different based on the specific restriction imposed on the host-side SoC architecture. For example, one level can block direct access to the system bus that can bypass physical memory protection units while another level allows such access.

These privilege levels present the minimum number of zones required for safe access to debug infrastructure in a secure SoC that has an integrated security engine. Based on

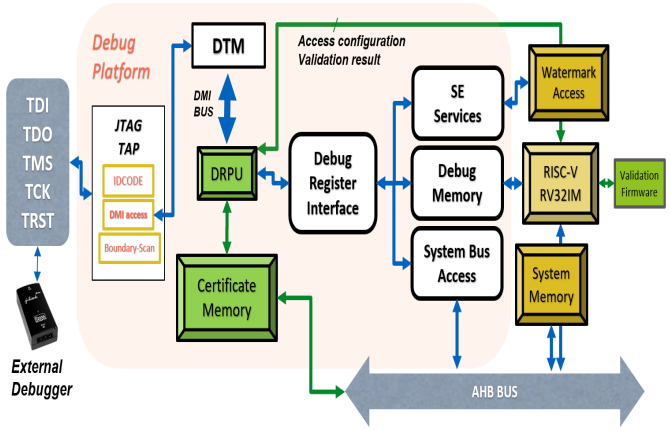

Fig. 4. RISC-V debug platform and with its connection to a simplified RISC-V SoC with a 32-bit core and a small memory for storing firmware.

the implementation, each level can be fragmented so we have a finer grain of policy enforcement.

IV. Results and Contributions

RISC-V, as a rapidly adopted open-source architecture, exposes significant security risks through its conventional debug interface. Unlike proprietary solutions like ARM's SDC-600, RISC-V lacks built-in secure debug, prompting the need for additional IPs to enhance its security without altering the existing specification.

A. RISC-V Debug

The RISC-V debug architecture provides a flexible and feature-rich interface for hardware-assisted debugging of RISC-V processors. At full functionality, it enables reading and writing of all hardware threads' (harts) registers, including control and status registers (CSRs). Memory access can be performed either from the hart's perspective or directly via a master port on the system bus. The architecture supports independent debugging of each hart and allows automatic discovery of system configuration, requiring minimal manual setup. Debugging can begin from the first instruction using the debug reset interface, with support for breakpoints, single-stepping, and other granular control mechanisms. Optional features include halting/resuming subsets of harts, executing custom instructions on halted harts, non-intrusive register access, and programmable triggers based on PC, memory address, or data [15].

A simplified RISC-V SoC with its debug architecture is illustrated in Fig.4, excluding the green-shaded contributions of this paper. External debuggers typically connect via JTAG (IEEE 1149)[4], which interfaces with the Debug Transport Module (DTM). The DTM drives the Debug Module Interface (DMI), connecting to the Debug Module (DM), which exposes the full debug capability via a register-based interface. The DM supports three main categories: (1) Run/Halt control for pausing/resuming harts, (2) Abstract commands and program buffers for executing arbitrary instructions and accessing registers, and (3) System bus access for interacting with SoC peripherals. Access to these features is managed through read/write operations on specific debug module registers, such as dmcontrol (hart selection), command and data0–11 (instruction and return data), and sbaddress/sbdata (system bus interface). This centralized register interface serves as an ideal monitoring point for enforcing fine-grained access control over debug operations. For instance, write protection on sbaddress can restrict unauthorized access to SE-connected buses [15].

TABLE I
Power and Area Overhead of Secure Debug IPs

	Secure Debug	RISCV CVA6 Core	RISCV Debug	Overhead for debug	overall overhead
Area (nmm)	2608.24	245871.38	49440.84	5%	0.8%
Dynamic Power (uW)	311.19	8861.1	2311.8	13%	2%

B. Debug Register Protection Unit (DRPU)

With the identification of the debug register interface as a critical enforcement point, we propose a hardware module—Debug Register Protection Unit (DRPU)—to implement and enforce security policies for secure debug. The DRPU is integrated with the RISC-V debug architecture and monitors accesses to the debug module's register interface. The DRPU has three primary functions: (1) Certificate Channel Support: We introduce a new authentication opcode alongside the conventional DMI read/write operations. This opcode redirects data from the JTAG register to a certificate buffer memory. The debugger certificate is shifted into the system using this opcode, and the DRPU manages access to the buffer based on address decoding. (2) Access Control Enforcement: After certificate validation, access permissions are configured via firmware running on the secure processor. The DRPU enforces these restrictions during conventional debug operations. (3) Status Reporting: The DRPU provides real-time feedback to the debugger, indicating authentication status and the state of the debug infrastructure. In our prototype, we implemented three core policies: blocking access to SE-related harts, restricting system bus access to SE-protected addresses, and preventing external access to watermark-related registers. Developers can extend this model to enforce additional policies tailored to their SoC's threat model. Our implementation is based on the open-source CVA6 core, a 6-stage, single-issue, in-order RISC-V CPU (RV32IM). The certificate validation is handled by secure firmware within the SE. The minimal area, power, and timing overhead introduced by the DRPU and certificate buffer are summarized in Table IV-A, demonstrating the efficiency of our secure debug solution.

V. Conclusion

This work addresses the critical challenge of secure hardware-assisted debugging in SoCs that integrate Security Engines (SEs). While debug interfaces are essential for development and diagnostics, they pose significant security risks if left unprotected—especially in open architectures like RISC-V. We presented a layered framework that uses certificate-based authentication and fine-grained privilege control, enforced through a Debug Register Protection Unit (DRPU) integrated into the RISC-V debug path. Our hardware prototype demonstrates the feasibility of enforcing access control without modifying the RISC-V debug specification. Experimental results show minimal area and power overhead, validating the practicality of our approach.

References

[1] Sk Subidh Ali, Ozgur Sinanoglu, and Ramesh Karri. "Test-mode-only scan attack using the boundary scan chain". In: *2014 19th IEEE European Test Symposium (ETS)*. 2014.

[2] ARM Limited. *Security technology: Building a secure system using TrustZone® technology*. Online. 2008. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.prd29-genc-009492c/index.html.

[3] Subodha Charles, Vincent Bindschaedler, and Prabhat Mishra. "Digital Watermarking for Detecting Malicious Intellectual Property Cores in NoC Architectures". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2022).

[4] "IEEE Standard for Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (2013).

[5] Intel. *Intel® Software Guard Extensions programming reference*. Online. Accessed on: Aug. 9, 2019. 2014. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[6] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. "Trusted Execution Environments: Properties, Applications, and Challenges". In: *IEEE Security Privacy* (2020).

[7] Sudeendra Kumar et al. "PUF-based secure test wrapper for SoC testing". In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2018, pp. 672–677.

[8] Vinay BY Kumar et al. "Towards designing a secure RISC-V system-on-chip: ITUS". In: *Journal of Hardware and Systems Security* (2020).

[9] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. 2020.

[10] Kuen-Jong Lee, Zheng-Yao Lu, and Shih-Chun Yeh. "A Secure JTAG Wrapper for SoC Testing and Debugging". In: *IEEE Access* (2022).

[11] Zhenyu Ning and Fengwei Zhang. "Understanding the Security of ARM Debugging Features". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[12] Xuanle Ren et al. "IC Protection Against JTAG-Based Attacks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019).

[13] Md Sami Ul Islam Sami et al. "Invited: End-to-End Secure SoC Lifecycle Management". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021.

[14] Alan Sguigna. "Mitigating JTAG as an Attack Surface". In: *2019 IEEE AUTOTESTCON*. 2019.

[15] "SiFive, RISC-V External Debug Support 0.13, 2018." In: ().

[16] Emanuele Valea et al. "A Survey on Security Threats and Countermeasures in IEEE Test Standards". In: *IEEE Design & Test* (2019).

[17] Emanuele Valea et al. "Encryption-Based Secure JTAG". In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. 2019.

[18] Sebastian Vasile, David Oswald, and Tom Chothia. "Breaking all the things—A systematic survey of firmware extraction techniques for IoT devices". In: Springer. 2019.

[19] Weizheng Wang et al. "Ensuring Cryptography Chips Security by Preventing Scan-Based Side-Channel Attacks With Improved DFT Architecture". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2022).