SRACARE: Secure Remote Attestation with Code Authentication and Resilience Engine

Avani Dave, Nilanjan Banerjee and Chintan Patel
Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County

***Abstract*—Recent technological advancements have enabled proliferated use of small embedded and IoT devices for collecting, processing, and transferring the security-critical information and user data. This exponential use has acted as a catalyst in the recent growth of sophisticated attacks such as the replay, man-in-the-middle, and malicious code modification to slink, leak, tweak or exploit the security-critical information in malevolent activities. Therefore, secure communication and software state assurance (at run-time and boot-time) of the device has emerged as open security problems. Furthermore, these devices need to have an appropriate recovery mechanism to bring them back to the known-good operational state. Previous researchers have demonstrated independent methods for attack detection and safeguard. However, the majority of them lack in providing onboard system recovery and secure communication techniques. To bridge this gap, this manuscript proposes *SRACARE* - a framework that utilizes the custom lightweight, secure communication protocol that performs remote/local attestation, and secure boot with an onboard resilience recovery mechanism to protect the devices from the above-mentioned attacks. The prototype employs an efficient lightweight, low-power 32-bit RISC-V processor, secure communication protocol, code authentication, and resilience engine running on the Artix 7 Field Programmable Gate Array (FPGA) board. This work presents the performance evaluation and state-of-the-art comparison results, which shows promising resilience to attacks and demonstrate the novel protection mechanism with onboard recovery. The framework achieves these with only 8% performance overhead and a very small increase in hardware-software footprint.**

***Index Terms*—Secure-boot, Remote attestation, Embedded System Architecture, IoT devices, Attack resilience, Fault Tolerant and Trusted Embedded Systems, Intelligent Embedded Systems**

I. Introduction

The recent technological advancement has catastrophically increased the utilization of small embedded and IoT devices in applications ranging from industrial control systems, vehicular systems, and home automation systems. Attack [1] has demonstrated that the software on these devices can be compromised even when powered off. Remote malware attacks such as Stuxnet [2] and Jeep [3] can modify the firmware or software of the device. Other attacks such as man-in-the-middle [4], record & replay [5] have shown that the security-critical information of the device can be leaked, modified, and utilized for malevolent activities. Attacks such as Denial of Service (DoS) [6] can flood the communication interface of an application to disrupt or damage its normal operation. Therefore, secure communication and software state assurance (at run-time and boot-time) has become paramount essential for the system's security assurance. Unfortunately, the small embedded and IoT systems are computationally weak and do not have in-built security and integrity checking primitives. Hence, they are an active substrate for cyber-attacks that violates software integrity or use the leaked critical information in malicious activities.

Secure boot is a process of measuring the **boot-time** integrity and authenticity of the software running on the device. It assures that the device boots up with an untampered and authorized software provided by a legitimate vendor. Thus, a secure boot process becomes a critical step in the device firmware and software security at boot-time. The Remote Attestation (RA) is a popular method of detecting the malicious code presence on the device. RA is a client-server protocol between an untrusted Prover (P_r) and a remote trusted Verifier (V_r) devices. The V_r requests the P_r for the proof of integrity and/or authenticity of the current state of the device software at run-time, P_r performs the appropriate checks and sends the report to the V_r. The V_r validates the integrity and authenticity of the current state of the software on the P_r. Previous work has demonstrated RA's utility for software updates [7] and deletion [8]. The conventional secure boot and RA systems often stop or reset the device upon detecting the malevolent code presence. The device requires the code reflash to restore it to the normal operational state. These brings the requirement of recovery/reflash logic.

While conventional RA systems provide run-time detection of corrupted software state, it suffers from the following limitations:

- It does not provide an efficient recovery mechanism.
- It is used for detecting the integrity and authenticity of the system's software state at run-time and not at boot-time. The system needs a secure boot for boot-time software state assurance.
- The majority of conventional RA systems did not have the secure communication protocol between the RA devices, which can result in leakage of cryptographic keys, information, or reports and potential misuse by the adversaries.

The conventional secure boot and RA devices perform the recovery either by reflashing the application code memory over-the-air or manually. The former method is prone to man-in-the-middle [4] and replay [5] type of attacks. A smart attacker can corrupt the networking stack to fail the over-the-air code reflash. These necessitates manual intervention for the recovery of the device. Currently used embedded and IoT devices have a wide variety of applications ranging from ships, industrial plants, CCTV cameras, and distributed network sensors (such as dust nodes). These applications require the placement of these devices into the locations that are relatively hard to access for performing device maintenance. Therefore, those devices require an **onboard recovery mechanism** to reduce the downtime and maintenance cost. Recent implementations such as [9] and [10] attempt to provide recovery methods to the affected devices. However, they lack in providing either the secure boot or RA implementation.

To address these limitations, the proposed work presents *SRACARE* a framework that uses a custom communication protocol to measure (run-time and boot-time) integrity and authenticity of the device's software. It also presents a novel resilience and onboard recovery method to protect the device from the malicious code modification attacks.

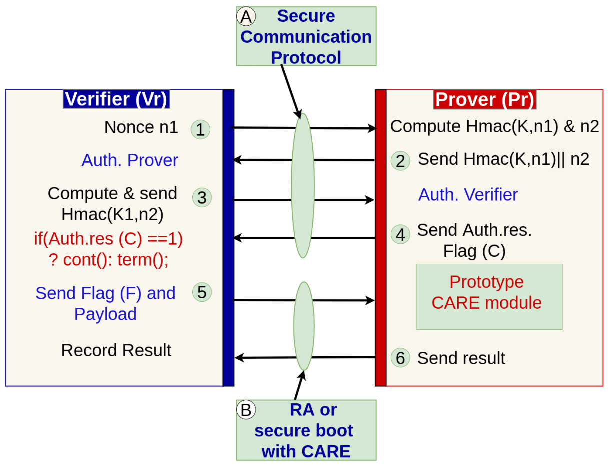

Fig. 1. Highlights the proposed *SRACARE* system design flow and key contributions. It represents the lightweight authenticated secure communication protocol, and a new remote attestation and secure boot architecture using custom *CARE* module.

Fig 1 provides a high-level design overview of the proposed framework. The *SRACARE* system operation is categorized into two sub-tasks: Ⓐ Authenticating the trusted V_r and un-trusted P_r using lightweight, secure communication protocol and Ⓑ Based on the result of task Ⓐ, it performs either remote (run-time) or secure-boot (boot-time) attestation with onboard recovery by using the prototype *CARE* module. It sends the final computed result to the V_r, as depicted in step ⑥ of Fig 1. The authentication of the V_r and P_r are performed by sending nonces both ways and computing HMAC (Hash based Message Authentication Code) using a prototyped communication protocol (as shown in steps ① to ④). The novelty of this approach resides in the lightweight design of the nonce (n2) generation mechanism at the P_r side, without requiring the resource heavy nonce generation techniques such as True Random Number Generator (TRNG), details of which are covered in section §IV-A and section §IV-B. If the authentication check fails the *SRACARE* system sends Flag (C==0) to the V_r as an attack indicator and V_r closes the communication between the devices. Upon authentication passing, the P_r sends Flag (C==1) and the V_r continues the next step by sending Flag (F) and associated payload to the P_r device. The P_r system performs either secure boot with *CARE* or remote attestation based on the value of Flag (F) (as shown in steps ⑤ and ⑥). The (highlighted) prototype *CARE* module provides novel malicious code modification attacks detection, protection and onboard recovery mechanism for small embedded and IoT devices.

Research Contributions: The design and implementation of the proposed *SRACARE* framework presents the following research contributions:

- **Secure Communication Protocol:** It demonstrates the prototype implementation of a lightweight secure authenticated communication protocol between the trusted V_r and untrusted P_r.
- **Nonce Generation Technique:** It provides a lightweight novel nonce (n2) generation technique that ensures freshness for the P_r device without requiring substantial system resources.
- **Remote Attestation Tools:** It provides the tools to verify the integrity and authenticity of the state of the software on the device at run-time by the remote V_r.
- **Secure Boot:** *SRACARE* presents lightweight, secure boot (boot-time attestation) architecture prototype for RISC-V based small embedded and IoT devices.
- **Resilience Engine:** It demonstrates the first implementation of the novel resilience engine that provides onboard recovery and protection method to recover the P_r device after malicious code modification attack.
- **Prototype *SRACARE* Implementation:** It presents the prototype implementation of *S*RACARE framework on FPGA, which can be used as a standalone microcontroller or as a secure boot co-processor such as the Trusted Platform Module (TPM) in large devices.

II. Background and Related Work

This section defines essential concepts used and referenced by the proposed framework, followed by exploring related state-of-the-art works. Measured boot [11] is a process of verifying the integrity of the software running on a system. Authenticated boot [12] verifies that the software running on the system is coming from an authorized vendor. The majority of the conventional secure boot and RA techniques perform either measured boot or authenticated boot, and very few support both. The onboard recovery and secure RA communication will be the requirement of the next-generation embedded and IoT devices, as discussed in section §I. Previous researchers have implemented secure boot and/or RA techniques that can be classified into hardware, software, and hybrid approaches:

Hardware-Based: One of the popular methods of secure boot and RA uses a discrete co-processor called the Trusted Platform Module (TPM) [13] recommended by the Trusted Computing Group (TCG). TPM has a special purpose registers called Platform Configuration Registers (PCRs),

which cannot be overwritten. However, it can only be extended by hashing the software measurements together with the previous PCR values. The TPM can sign the PCRs with a private attestation key to generate a piece of attestation evidence. The TPM provides hardware root-of-trust. However, it is not suitable for small embedded or IoT devices due to space, size, and cost constraints. Some researchers have used the Trusted Execution Environment (TEE) [14], Keystone [15], or proprietary implementation of Arm TrustZone [16] for runtime attestation. TrustZone uses two virtual processors called the secure and normal world to enforce the hardware-based isolation. Microsoft's fTPM [17] provides a use-case of TrustZone for secure boot and attestation. Intel's SGX [18] provides instruction and memory access features that can be used to instantiate protected containers referred to as enclaves by using special instructions and processor extensions. TrustZone and TEE increases the design complexity and cost associated with exclusive licensing. Other secure boot architectures have used complex Unified Extensible Firmware Interface (UEFI) [19] design or utilize heavy resources [20], which makes them unsuitable for small embedded and IoT devices.
Software-Based: Researchers have used simulated or software TPM implementations like simTPM [21] or IBM's software TPM [22] for secure boot and RA. RISC-V based implementation Sanctum [23] uses software-based enclaves for attestation. The implementation presented in [24] has used cryptographic software core and hash engine for attestation.
Hybrid: SMART [25] is a dynamic root-of-trust architecture for low-end devices at runtime. SPM/Sancus: SPM [26] and Sancus [27] present a security architecture that provides isolation of software modules using additional CPU instructions. TrustLite/TyTAN: TrustLite [28] and its successor TyTAN [29] provide flexible, hardware-enforced isolation of software modules using Execution-Aware Memory Protection Unit (EA-MPU). Google's latest implementation Opentitan [30] provides a secure boot based hardware root of trust. Other RISC-V based secure boot and attestation architecture Shakti-T [31] uses base and bounds concept to secure the pointer's access to the valid memory regions. The existing secure boot and RA architectures are complex [12], require more resources [21], [14], [16], or have been compromised by attacks such as [32] and [33].

Moreover, none of the available solutions provide the recovery and they use conventional message authentication protocols, which are resource heavy and not suitable for small embedded and IoT devices. Recently implementation Healed [9] presents the first recovery mechanism using Merkle Hash Tree (MHT). It assumes that at least one node in the network is untempered, and firmware of that node is used to reflash the corrupted node. Another implementation [10] uses a method of putting software receiver-transmitter code into trusted ROM to connect the device to a recovery server. It requires additional ROM storage, processor, and internal communication bus to be part of Trusted Computing Base (TCB). The proposed *SRACARE* method uses a different approach of storing the recovery data in a secure backup ROM, and it does not require processor and internal communication bus to be part of TCB to reduce the attack surface.

III. Security Model

A. Security Properties

SRACARE has derived eight (A1-A8) security properties (from [34]) for the secure boot and remote attestation system design, which are broadly classified into three main domains, namely: Secure Communication, Key Protection, and Safe Execution.
1) Secure Communication: The communication protocol between the V_r and P_r devices should be resilient to wiretapping and flooding types of attack. **A1. Eavesdrop Protection:** The devices should have wiretapping and eavesdropping protection to prevent attacks such as [35], [5], or [4] which can cause security-critical information leakage or misuse. **A2. Flooding Protection:** It also requires protection from attacks such as [6] DoS and [36] DDoS, which can fail the application.
2) Key Protection: The shared cryptographic secret key (K) and device secrets should not be exposed to the adversary and stored in protected memory with no unauthorized access. **A3. Key Confidentiality:** The secure key (K) must be stored in protected memory or ROM. **A4. Access Control Enforcement:** It requires proper access control policies to prevent unauthorized read-write access to the protected memory.
3) Safe Execution: The design should have an error-free, uninterrupted, and leak-proof implementation and execution of all the system modules. **A5. Correct Implementation:** The implementation of all the submodules should be untempered and correct. **A6 Atomicity:** Once triggered, the code execution should not be interrupted. **A7. Error Free Execution:** All the hardware (IPs) and software submodules should have error-free execution. **A8. Controlled Invocation:** The system requires proper code triggering and execution sequence, with no privilege escalation or interruption.

B. Adversary and Threat Model

The adversary:

- has full control over the firmware and software of the device.
- can perform an unauthorized read or write to the flash memory.
- can modify existing code by re-arranging, flipping the bits, buffer-over-flow, or fault injection attacks.
- cannot attack protected ROM.

The side-channel attack, physical access, damaging device, and control flow integrity attacks are out of scope for the proposed work.

C. Design Choices

SRACARE system incorporates the following design choices to satisfy the security properties discussed in subsection §III-A.

❶ **Secure Storage:** *SRACARE* uses separate ROM for storing the device information such vendor ID, Unique Device Identification (UUID), firmware revision, the symmetric cryptographic shared key (K) and extends the layout of the secure ROM to store a trusted recovery image.

❷ **Frame Data Structure and Internal Communication:** The conventional secure boot and RA systems uses an internal system bus for communication between the trusted and untrusted hardware-software modules. Attack [37] has demonstrated that it can boot the device through malicious hardware connected to the same interconnect bus. Therefore, the proposed framework uses a dedicated SPI bus for communication between the ROM, flash, and prototype *CARE* module. Moreover, the conventional system computes the digest over the entire firmware image and transfers the results via a universal interconnect bus. Since *SRACARE* uses the SPI bus, if the SPI tool's hardware or software gets corrupted, it can send occasionally corrupted bits or wrong data. Therefore, the proposed design divides the flash image into 1 KB chunks. Section §IV-B covers the details of the frame data structure. This design choice is used for Proof Of Concept (POC) implementation only, and the user can parameterize frame size to optimize the performance.

❸ **Code Integrity & Authentication (CA) Unit:** Most modern secure boot architecture uses two crypto-cores - one for the digest computation and others for code signing. After performing speed, resource, and cost evaluation of various crypto-cores such as AES, RSA, ECDSA, SHA3, SHA256, SHA384, and HMAC-SHA256, the design choice of using single hardware implementation of the crypto-core HMAC-SHA256 is made for digest computation and code authentication. This hardware reuse makes the proposed solution lightweight and suitable for the targeted small embedded and IoT devices.

❹ **Resilience Engine (RE):** RE provides the ability to recover the device from memory modification attacks by reflashing the affected flash memory region using onboard recovery (backup) code, It also applies the access control policies to protect the device from future attacks.

❺ **Trusted Execution:** *SRACARE* provides trusted execution environment by isolating the TCB and processor, and using a dedicated SPI bus for internal communication.

❻ **Secure Communication Protocol:** *SRACARE* uses the proposed lightweight, secure communication protocol to satisfy the secure RA properties A1 and A2 from subsection §III-A by leveraging existing hardware resources and following steps 1 to 12 from Fig 2. It also demonstrates a custom nonce generation technique for small embedded and IoT devices.

IV. *SRACARE* Framework

A. System Design

Fig 2 shows the top-level design overview of the *SRACARE* framework, highlighted are two main security-enhancing techniques proposed by this work: 1) Lightweight, secure communication protocol and 2) Secure boot with *CARE* and RA architecture for the P_r device. The notations and definitions used for the communication are listed in Table I. *SRACARE* establishes lightweight, secure communication protocol between the trusted V_r and un-trusted P_r by following steps 1 to 12 from Fig 2, and the detailed working is explained in subsection §IV-B.

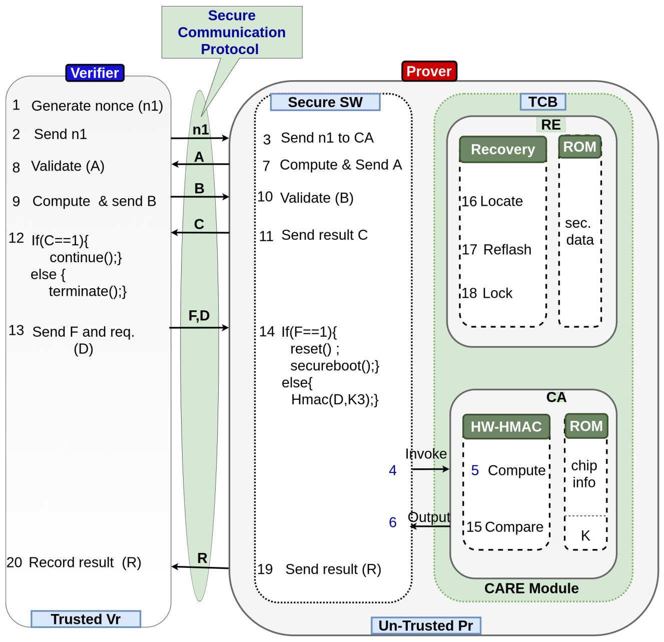

Fig. 2. Highlights the system design and key contributions of *SRACARE*: 1) Novel lightweight, secure authenticated communication protocol (steps 1 to 12), and 2) Secure boot with *CARE* and remote attestation architecture for the P_r device (steps 13 to 20)).

TABLE I
NOTATIONS AND DESCRIPTION

Notation	Description
n1	V_r's nonce for freshness

n2	P_r's nonce for freshness
	n2 = Hmac(K, T)
	T = hash(CHIP INFO.) $\oplus$ n1

K	Symmetric key for HMAC
Hmac(K, m)	H(($K' \oplus$ 0x5C5C) \|\| H(($K' \oplus$ 0x3636) || m))
A	A = Hmac(K, n1) $>>$ n2
B	B = Hmac(K_1, n2)
C	C is a true or false result of the validation of B.
D	D consists of parameters S_{addr} and L as payload for attestation
F	Reset Flag
S_{addr}	Start address of flash memory for hashing
L	Lenth of the memory region to be hashed
R	Final Result
K'	$\begin{cases} H(K) & K \textit{ is larger than the block size} \\ K & \textit{otherwise} \end{cases}$
m	Memory region to be attested, derived from S_{addr}, L
H	Cryptographic hash function
K'	Key derived from the secret key K
K_1	K_1= (Hmac(K, n1) $\oplus$ n1 $\oplus$ n2))
||	Denotes concatenation
$\oplus$	Denotes bitwise exclusive or (XOR)
CA	Code Authentication
RE	Resilience Engine
RA	Remote Attestation

The proposed secure communication protocol has two advantages over conventional authenticated communication protocols: (1) It authenticates both end devices (the P_r and V_r) in the communication and provides resilience from [4], [5], and [6] attacks. (2) It does not require additional computationally heavy system resources such as TRNG, Authenticated Encryption with Associated Data (AEAD), Elliptic Curve Digital Signature Algorithm (ECDSA) or complex Message Authentication Code (MAC) to satisfy A1 & A2 security properties listed in section §III-A. To

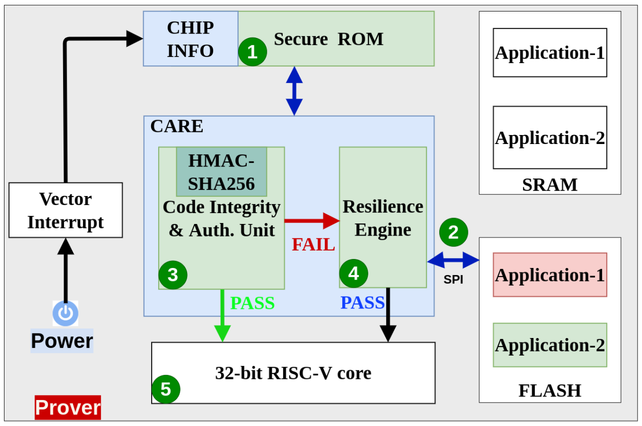

Fig. 3. Shows the architecture design of *SRACARE* based P_r system, highlighted are the key design modules. The pass arrows indicate that only the known good code will be allowed to be executed on the RISC-V core at any given time.

satisfy all the security properties from A3 to A8 discussed in section §III-A, *SRACARE* based P_r system follows design choices 1 to 5 listed in section §III-C, as highlighted in Fig 3. The P_r performs either the RA or secure boot with *CARE* by following steps 13 to 20 from Fig 2. The detailed working of the system is covered in section §IV-B.

B. System Operation

The working of *SRACARE* system is divided into four main steps: 1) Secure Communication Protocol, 2) Secure Boot, 3) Resilience and Recovery, and 4) Remote Attestation.

1) Secure Communication Protocol: The communication starts by the V_r sending nonce n1 to the P_r. The P_r computes Hmac(K, n1) using the crypto-core and generates n2 by performing Hmac(K, T).

$$n2 = Hmac(K, T)$$
$$T = hash(CI_{\text{start}}, 16) \;\; xor \;\; n1 \tag{1}$$

Where T is calculated by taking the hash of the first 16 Bytes of the secure chip information memory and xor it with the received value of n1 (for freshness). The chip information (CHIP INFO) memory consists of the device specific information such as device serial number, firmware version, and UUID as depicted in Fig 3. Term CI_{start} in equation 1 points the starting location of Chip Info (CI) memory. This novel approach gives unique n2 each time without requiring resource-heavy salt generation techniques such as TRNG. The P_r generates A = (Hmac(K, n1) >> n2) by appending n2 with Hmac(K,n1) and sends it to the V_r. The V_r validates the authenticity of the P_r by recomputing Hmac(K, n1) and matching it with the received value. The V_r derives the new secret key K_1, computes Hmac(K_1, n2), and sends the result to the P_r. The P_r follows the appropriate generation and validation steps to authenticate the V_r and sends the result Flag C (step 11 from Fig 2) to the V_r. *SRACARE* closes the POC UART connection (it can be Xbee or other) between the P_r and V_r devices when the V_r receives (C==0) (in step 12 from Fig 2), else it sends the Flag F defining the next action and associated payload to the P_r. **2) Secure Boot:** If the received Flag (F) is set (F==1), then the P_r calls system reset function and performs the secure boot with *CARE*. Note that steps 4 to 6 in Fig 2 are represented differently to denote that those steps will be part of both RA or secure boot. However, the sequence of execution will be different. As depicted in Fig 3, the secure boot sequence starts with the system power-on. It locates and executes the First Stage Boot Loader (FSBL) code from secure ROM to initialize the SPI and flash controllers, read chip information such as - device UUID, board version, and symmetric share key, and hand off the control to the second stage boot code called the bootstrap. The bootstrapping process divides the flash image into a 1 KB frame chunks and sends it one at a time to the host via SPI bus for integrity and authenticity check. Each frame consists of the header and associated payload, as indicated in Fig 4. The header contains the digest of the

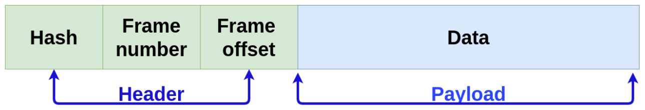

Fig. 4. Represents the frame data structure. The header contains the digest of the entire frame, frame number, and flash offset location. The payload contains corresponding data for each frame.

entire frame, frame number, and the flash offset location. The offset location is the flash memory offset location used for the frame reflashing. The payload contains the corresponding data for each frame. This work has leveraged Hash based Message Authentication Code's (HMAC) feature for signing (authenticating) the data and SHA256 feature for integrity check for each frame to reduce hardware footprint and cost. Secure boot follows steps 4-5-15-16-17-18-6 from Fig 2 for each frame, and upon digest mismatch detection, the P_r triggers the RE else the device will continue the normal boot process. **3) Resilience Engine:** RE follows steps 16-17-18 from Fig 2 to locate the affected memory region, reflash the corrupted flash memory region with the known good software code from secure ROM, and lock the unauthorized access to the flash region using Physical Memory Protection (PMP) mechanism of the RISC-V processor. **4) Remote Attestation:** If the received Flag (F==0) value is not set, the P_r performs remote attestation based on the payload provided by the V_r, which consists of the start location and the length of the information to be attested. The P_r follows steps 4-5-6 sequence from Fig 2 to compute the digest and it sends the report to the V_r (steps 19 and 20 from Fig 2).

V. Evaluation

This section describes the chain-of-trust theory, resource utilization, and performance analysis for each submodule in *SRACARE* framework design, and comparison with state-of-the-art solutions.

A. Chain-of-Trust

The work presented in [11] defines the secure boot process as a chain of many small layers of the boot codes executed in a specific sequence. It requires the boot process to follow two rules for the integrity measurement assurance: **1)** The integrity is checked for all the lower layers. **2)** Transitions to higher layers occurs-only after integrity checks on all lower layers are completed. The conventional secure boot systems measure the integrity of each stage (layer) of the boot code at the file level. The proposed work uses the same concept in a different context as it breaks down the entire image into 1 KB chunks (called frames) and measures code integrity and authenticity of each frame. Following equation represents the chain-of-trust for *SRACARE* system:

$$\begin{aligned} I_0 &= True \\ I_{i+1} &= I_i \quad \& \quad V_i(L_{i+1}) \end{aligned} \tag{2}$$

I_i and & are the boolean value representation of the integrity of frame i and AND operation respectively. The verification function associated with the i^{th} layer is represented by V_i. V_i takes the layer to verify as its only argument and returns boolean result. It performs a cryptographic hash of the frame and compares the result with stored digest value. As explained earlier, this work has divided boot flash data into 1 KB frame chunks, and hash digest is computed on each frame for verification. Therefore, the recurrence is represented by:

$$I_{i+1} = \begin{cases} I_{i=0} = True & for \;\; i = 0 \\ I_i \;\; \& \;\; V_i(L_{i+1}) & for \;\; i = 1, 2, 3...n \end{cases} \tag{3}$$

Here, I_0 is the trusted boot ROM code, and the integrity of ROM is taken as boolean true. n represents the number of frames of the flash image. The following equation 4 calculates the estimated increase in boot time ($T_\triangle$) for a secure boot with *SRACARE*.

$$T_\triangle = t_{\text{fm\_0}}(V_0(L_1)) \quad + t_{\text{fm}}(\sum_{i=1}^{n} V_i(L_{i+1})) \tag{4}$$

Where $t_{\text{fm\_0}}$ is the execution time for the first frame, and t_{fm} is the execution time for all remaining frames. The verification time includes the time required to compute and verify the message digest. By design, *SRACARE* first matches the frame number of the received frame and clears the flash region to reflash it with a trusted code. Therefore, the first frame processing requires more time ($t_{\text{fm\_0}}$) than the remaining frames.

B. Code Authentication (CA) Unit

The key component of the Code Authentication (CA) unit is crypto-core (HMAC-SHA256). To estimate the initial performance, power, and resource utilization, our evaluation setup uses both hardware and software implementation of the crypto-core running on the baremetal RISC-V processor to compute the digest of same data size of 256 Bytes. Table II illustrates a performance increase of 16x and 92% less power utilization while using the hardware HMAC-SHA256. Also, software implementation requires 3.6 KB additional secure storage and assumes the RISC-V core to be part of TCB. Including the processor in TCB is not a preferred design choice due to the processor related vulnerabilities. Therefore, a hardware-based crypto-core is selected for the CA unit.

TABLE II
Performance Analysis of CA on FPGA.

Performance Analysis of crypto-core on FPGA		
Parameters	Software	Hardware
Cycles (c)	**47033**	**2926**
Frequency (f)(MHz)	100	100
Block (b)	256	256
Throughput T(Mbps)	.54	8.74
Time (usec)	470.33	29.26
Energy Consumption (E)	197.06	12.25
Energy Efficiency	**92.68**	**0.358**

C. Resilience Engine (RE)

The Resilience Engine (RE) is implemented in software for the POC work. For the test application of 5.6 KB, it requires **61** additional lines of code (C language) in the secure boot code base and approximately 5 KB of additional secure ROM to store the golden recovery image data. The Resilience Engine (RE) requires 968 bytes of recovery data for every 1 KB of the flash image. The secure ROM size increases exponentially with the size of the application code used for recovery. To limit the size of the recovery data storage, the application developer can select the necessary code module for the recovery process or use a suitable compression technique, to bring the device to the bare minimum working state. Although, this feature is not implemented in the proposed work as the test application uses only 5 KB of additional ROM for recovery.

D. System Performance

For *SRACARE* system **timing analysis**, a test application of 5.6 KB is divided into six 1 KB chunks and the total time for the system boot-up with and without the proposed framework is calculated and presented in Table III. The framework uses equation (4) to calculate the total execution time T and equation (5) for time difference $D_\triangle$. As seen from Table III, the time difference in the total time $D_\triangle$ = .529 milliseconds indicates that proposed *SRACARE* architecture requires 8 percent extra boot-time, 5 KB extended ROM and increases bootstrap code by 61 lines, which are insignificant in comparison to the level of security, onboard recovery, and resilience it provides.

TABLE III
TIMING ANALYSIS FOR BOOTSTRAP.

Timing Analysis of secure bootstrap on FPGA		
Parameters	Without	With
Cycles req. for the first frame (c)	553611	576083
Cycles (rest of frames)(c)	103330	133790
Total Cycles	**656941**	**709873**
Frequency (f)(MHz)	100	100
Time (T) (usec)	**6569.41**	**7098.73**
Time difference $D_\triangle$ = **529.32** usec which is 8% more than without secure boot		

$$D_\triangle = Time(with\textit{SRACARE}) - Time(without\textit{SRACARE}) \tag{5}$$

E. Resource Utilization

Inspired from the googles opentitan [30], the prototype implementation uses the Ibex RISC-V core for the RTL design.

TABLE IV
HARDWARE RESOURCE UTILIZATION.

FPGA Hardware Resource Utilization Report				
Parameters	*SRACARE*	HMAC	Ibex	$\%Util.$
Slice LUTs	**24249**	**2807**	3581	18
LUT as logic	24081	2807	3581	18
LUT as DRAM	168	0	8	1
Slice Registers	19586	2312	2559	7
Register as FF	19581	2312	2559	7
Register as Latch	5	0	0	<1
F7 Muxes	1407	71	265	2
F8 Muxes	196	29	0	1

1) **Hardware Resource Utilization on FPGA:** Table IV depicts resource utilization of (crypto-core) HMAC-SHA256, Ibex core, complete *SRACARE* design, and percentage utilization of available hardware resource on Artix 7 FPGA. The crypto-core uses 3x less area and operates at 2x faster speed compared to other implementation from [38].

2) **Software Resource Utilization:** The POC work has implemented the RE submodule and secure communication protocol in software. It requires 61 additional lines of (C language) code for RE (which includes cycle calculation and analysis code for Ibex) and 5 KB of extended ROM storage.It requires 108 additional lines of code (C language) on the P_r side for secure RA (UART) protocol and novel n2 generation logic implementation. *SRACARE* increases the total software code base by 10%.

F. Comparison with the State-of-the-art Solutions

Since RISC-V is a relatively new architecture, authors of this work did not found any **secure boot with resilience or recovery** implementation for state-of-the-art comparison. Therefore, this work compares the proposed *SRACARE* framework with other state-of-the-art secure boot and RA architectures targeting RISC-V based embedded and IoT systems. The comparison focuses on both qualitative and quantitative analysis. Table V illustrates the qualitative comparison.

TABLE V
QUALITATIVE COMPARISON BETWEEN SECURE BOOT/RA TECHNIQUES TARGETING LIGHTWEIGHT EMBEDDED DEVICES.

Parameters	*SRACARE*	Healed	Ref. [10]	Ref. [20]	Sanctum
Design Type	Hybrid	SW	Hybrid	HW	Hybrid
Secure RA	yes	no	no	no	yes
Detection	yes	yes	yes	yes	yes
Protection	yes	no	yes	yes	yes
Recovery	yes	yes	yes	partial	no
Secure boot	yes	no	no	yes	yes

1) **Qualitative Comparison:** Table V shows that all techniques provide memory attack detection methods, and only *SRACARE*, Healed, and [10] provides resilience and recovery techniques. Sanctum [23] uses hybrid secure enclaves for code execution and RA. [20] uses hardware-based memory attack detection and protection method. Healed [9] implements RA with recovery using Markle Hash Tree (MHT). It assumes that at least one device on the network is untampered, and the code of that device can be used for recovery. [10] provides a recovery method using a trusted ROM to store transmitter and receiver code for associating with a recovery server. Both [23] and [20] uses Rocket chip RISC-V core with OS support and complex designing. However, since [20] provides partial recovery support it is close candidate for *SRACARE* comparison.

TABLE VI
QUANTITATIVE COMPARISON WITH STATE-OF-THE-ART SECURE BOOT / REMORT ATTESTATION (RA) TECHNIQUE FOR RISC-V.

Parameters	*SRACARE*	HMAC	CAU's ECDSA only
Slice LUTs	24249	2807	27170
LUT as logic	24081	2807	26450
LUT as DRAM	168	0	720
Slice Registers	19586	2312	6722
Register as FF	19581	2312	6722
Register as Latch	5	0	0
Multiplexer			
F7 Muxes	1407	71	684
F8 Muxes	196	29	0

2) **Quantitative Comparison:** Table VI shows that [20] requires 27170 slice LUTs for hardware crypto-core module (ECDSA) implementation, which is **14x** larger than *SRACARE*'s crypto-core hardware requirement. Furthermore, [20] requires two 64 bit the RISC-V cores for Trusted Execution Environment (TEE) implementation, hardware SHA3 for hashing, and configurable LFSR-based Physical Unclonable Function (CoLPUF) for key generation, ECDSA core for asymmetric signing, boot sequencer, and key management unit. These make [20] a resource-heavy solution and unsuitable for small embedded and IoT devices. The comparison of asymmetric and symmetric cryptographic hardware requirements provide an initial estimation of the overall hardware overhead requirements.

VI. CONCLUSION

SRACARE demonstrates the POC framework that performs the run-time and boot-time integrity and authenticity checks, secure boot with onboard recovery, and remote attestation on the RISC-V based small embedded and IoT devices. It implements the novel lightweight, secure authenticated communication protocol. It provides the tools for the detection, protection and recovery from malicious code modification attacks. The experimental results show good protection against malicious code modification attacks, with only 8% execution time overhead and a tiny increase in resource footprint.

References

[1] A. Furtak, Y. Bulygin, O. Bazhaniuk, J. Loucaides, A. Matrosov, and M. Gorobet, "Bios and secure boot attacks uncovered," 2014.

[2] J. Vijayan, "Stuxnet renews power grid security concerns," https://www.computerworld.com/article/2519574/stuxnet-renews-power-grid-security-concerns.html, June 2010.

[3] D. Schneider, "Jeep hacking 101," http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.html, 2015.

[4] "Man-in-the-middle Attack," https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[5] "Replay Attack," https://en.wikipedia.org/wiki/Replay_attack.

[6] DoS Attacks, "What is a denial of service attack (DoS) ?" https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos, 2017.

[7] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *WiSe '06*, 2006.

[8] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *ESORICS*, 2010.

[9] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik, "Healed: Healing and attestation for low-end embedded devices," in *Financial Cryptography*, 2019.

[10] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *ESORICS*, 2010.

[11] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, May 1997, pp. 65–71.

[12] B. R. Richard Wilkins, "Uefi secure boot in modern computer security solutions," https://uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2019.pdf, 2019.

[13] "TPM wiki," https://en.wikipedia.org/wiki/Trusted_Platform_Module, 2010.

[14] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug 2015, pp. 57–64.

[15] D. Lee, D. Kohlbrenner, S. Shinde, D. X. Song, and K. Asanovic, "Keystone: A framework for architecting tees," *ArXiv*, vol. abs/1907.10119, 2019.

[16] H. Jiang, R. Chang, L. Ren, and W. Dong, "Implementing a arm-based secure boot scheme for the isolated execution environment," in *2017 13th International Conference on Computational Intelligence and Security (CIS)*, Dec 2017, pp. 336–340.

[17] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "ftpm: A software-only implementation of a TPM chip," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 841–856. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[19] NSA Cyber Report, "UEFI DEFENSIVE PRACTICES GUIDANCE," https://www.nsa.gov/Portals/70/documents/what-we-do/cybersecurity/professional-resources/ctr-uefi-defensive-practices-guidance.pdf, 2017.

[20] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, "Lightweight secure-boot architecture for risc-v system-on-chip," in *20th International Symposium on Quality Electronic Design (ISQED)*, March 2019, pp. 216–223.

[21] D. Chakraborty, L. Hanzlik, and S. Bugiel, "simtpm: User-centric TPM for mobile devices," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 533–550. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chakraborty

[22] Goldman, Ken, "IBM-ACS," https://sourceforge.net/projects/ibmtpm20acs/, 2017.

[23] I. Lebedev, K. Hogan, and S. Devadas, "Invited paper: Secure boot and remote attestation in the sanctum processor," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 46–60.

[24] S. Hristozov, J. Heyszl, S. Wagner, and G. Sigl, "Practical runtime attestation for tiny iot devices," 2018.

[25] M. M. Wong, J. Haj-Yahya, and A. Chattopadhyay, "Smarts: secure memory assurance of risc-v trusted soc," 06 2018, pp. 1–8.

[26] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *SecureComm*, 2010.

[27] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *USENIX Security Symposium*, 2013.

[28] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592824

[29] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: Tiny trust anchor for tiny devices," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[30] "Opentitan: Secure boot SoC design," https://opentitan.org/, 2019.

[31] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *In Proceedings of the Hardware and Architectural Support for Security and Privacy. ACM*, 2017.

[32] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-induced fault injection on smartphone bypassing the secure boot," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sep. 2017, pp. 41–48.

[33] A. Cui and R. Housley, "Badfet: Defeating modern secure boot using second-order pulsed electromagnetic fault injection," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, ser. WOOT'17. USA: USENIX Association, 2017, p. 3.

[34] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1429–1446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/de-oliveira-nunes

[35] H. valia, "Eavesdropping attack- ensure they don't listen to you," https://www.ilantus.com/blog/eavesdropping-attack-ensure-they-dont-listen-to-you/, 2019.

[36] "Flooding," https://www.webopedia.com/TERM/F/Flooding.html, 2019.

[37] N. Jacob, J. Heyszl, A. Zankl, C. Rolfes, and G. Sigl, "How to break secure boot on fpga socs through malicious hardware," *IACR Cryptology ePrint Archive*, vol. 2017, p. 625, 2017.

[38] M. Juliato and C. H. Gebotys, "Fpga implementation of an hmac processor based on the sha-2 family of hash functions," 2011.