FVCARE: Formal Verification of Security Primitives in Resilient Embedded SoCs

Avani Dave

Student Member, IEEE, Nilanjan Banerjee, Member, IEEE and Chintan Patel, Member, IEEE

Abstract: ***With the increased utilization, the small embedded and IoT devices have become an attractive target for sophisticated attacks that can exploit the device's security-critical information and data in malevolent activities. Secure boot and Remote Attestation (RA) techniques verifies the integrity of the device's software state at boot-time and runtime. Correct implementation and formal verification of these security primitives provide strong security guarantees and enhance user confidence. The formal verification of these security primitives is considered challenging, as it involves complex hardware- software interactions, semantics gaps and requires bit-precise reasoning. To address these challenges, this paper presents FVCARE an end-to-end system co-verification framework. It also defines the security properties for resilient small embedded systems. FVCARE divides the end-to-end system co-verification problem into two modules: 1) verifying the (bit precise) initial system settings, registers, and access control policies by hardware verification techniques, and 2) verifying the system specification, security properties, and functional correctness using source-level software abstraction of the hardware. The evaluation of proposed techniques on SRACARE based systems demonstrates its efficacy in security co-verification.***

Keywords: secure boot, formal verification, resilient system, onboard recovery, attack resilient system, small embedded systems.

1. Introduction

The utilization of small embedded and IoT devices has increase multi-fold in recent times for collecting, processing, and transferring security-critical information and user data. It has also enabled sophisticated attackers such as [1]–[5] to leak, tweek, slink or exploit the security-critical information of the device for use in malevolent activities. Denial of Service (DoS) [6] can flood the communication interface of an application and disrupt the normal operation. Therefore, software state assurance (at run-time and boot-time) and secure communication have become essential building blocks for device security. Security primitives such as 1) secure boot measures integrity and authenticity of the software state of the device at boot-time. 2) Remote Attestation (RA) is a client-server security service, which uses a trusted third-party verifier (Vr) to send the integrity verification request to an un-trusted prover (Pr) device at runtime. The Pr computes the digest and sends the report to the Vr. Therefore, if correctly implemented, these security primitives provide a strong security guaranty about the software state of the device. Formal verification techniques are used to verify that the system posses the correct specification and security properties.

Some of the currently available verification techniques require manual inspection or hacking skills [7], [8] and they can be repeated, scaled, or completely automated. Furthermore, they are to miss the bugs as manual involvement. Other existing formal verification approaches can be broadly classified in two categories: 1) Representing the firmware code in hardware by instruction level abstraction or by compiling the firmware as assembly code, and using hardware verification tools such as [9]–[11] for subsequent analysis.

2) representing the abstraction of hardware as software and using software verification tools such as [12]–[15] to verify the necessary security properties. The former approach uses the complex instruction-level abstraction process that makes it ISA specific and difficult to scale. The latter approach focuses on abstracting security-specific hardware features in software.

Formal verification of security primitives such as secure boot and RA is considered a challenging problem, as it involves multiple complex hardware-software interactions. For example, in the case of a hybrid SRACARE based system (discussed in section IV), it initializes a set of hardware registers during system boot-up and applies access control policies. The verification technique needs to verify appropriate hardware registers setting (along with other firmware software features), which cannot be verified by software abstractions. Furthermore, currently available formal verification techniques use bounded model checking (BMC) [] only and do not cover all system specifications, security properties edge cases. It also lacks in providing co-verification techniques of modules interaction as discussed in subsection III., Therefore, to bridge this gap, This paper presents hardware-firmware co-verification framework FVCARE. The formal co-verification process in FVCARE is com- presses of 1) defining the system using a suitable mechanical model, 2) identifying and documenting the desired system properties in a succinct and intelligible way, and 3) providing proof that set system properties are satisfied. For providing proof of step (3)) FVCARE framework divides end-to- end system verification tasks into two categories: First, it uses automated hardware formal verification technique similar to that of vrased [16]. Secondly, it uses the abstraction of hardware representation in software techniques for performing not only bounded model checking but also assertion and weakness prediction checking. It uses modular plugins with Frama-C [17] tool for software-based design specifications and security properties formal verification.

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

- **Design Specifications & Challenges:** It defines the secure system design specification, Hardware (Hw), and Firmware (Fw) interactions during the authentication, secure boot, and RA computation. It also highlights the Hw-Fw co-verification challenges.
- **Defines Security Properties:** It defines the security properties for SRACARE based small embedded de- vice.
- **Formal (Hw-Fw) Co-Verification Framework:** It demonstrates the practicality of security properties specific hardware abstraction in software. It also performs formal verification using Frama-C [17] tool. Frama-C tool with three new plugins provides Weakness Pre- diction (WP), Value (assertions), and Linear Temporal Logic (LTL) specifications checking.
- **Formal Hw Verification:** It presents a formal hard- ware verification approach by converting system verilog hardware modules (for specific properties checking only) to SMV using Verilog2SMV. It verifies the specific security property using NuSMV [18] tool.

By combining all these, FVCARE presents the first formal co-verification framework to verify the security primitives and properties of complex firmware codes in a small embedded System on Chip (SoC) (example: SRACARE based system).

Organization

Section II covers the background, discusses the design challenges for formal verification framework, presents related work and security properties. Section III presents targeted system design, operation, adversarial model, and scope of verification. It is followed by section IV, covering the formal verification methodologies, discussing the hardware- software verification approaches used by FVCARE. Section V provides the evaluation summary of the FVCARE framework by sharing verification results and findings. FVCARE evaluates the state-of-the-art system design approaches and formal verification techniques. Section VI provides the concluding remarks for formal co-verification work of *SRACARE* based SoC design.

2. Background & Related Work

This section provides a brief overview of the background and related work of formal verification techniques.

2.1 Background

Although previous implementations of secure boot [19]–[26] and RA [16], [27], [28], have provided strong security guarantees about the software state of the device, the majority of them lack in providing prevention or recovery techniques, the device will be kept in hang or un-operational state upon detection of malicious code modification attacks, the device needs code reflash, which can be done by manually or over-the-air code reflash conventionally. In the event of a smart attacker corrupting the networking stack, over- the-air code reflash becomes unsuitable. Often manual code reflash becomes not feasible due to placement of the targeted devices in applications such as home security cameras, smart controllers in automotive, aviation, or industrial systems. This necessitates some form of onboard recovery techniques as represented by CARE [29]. Recent work presented in SRACARE [30] extends CARE [29] by enabling RA and secure communication. Therefore, FVCARE has selected recent SRACARE based secure RA with onboard recovery system as shown in fig 1 for end-to-end formal verification. The high-level system operation can be summarized in two Steps:1) authentication of Vr and Pr devices using a secure communication protocol (steps 1 to 4) and 2) performs either remote attestation (run-time) or secure-boot (boot- time) with onboard recovery, depending on the result of step 1) Upon authentication failure, Pr sends a flag (C=0), and Vr closes the communication. Pr sends Flag (C=1) when authentication passes. Vr sends Flag (F) and payload to the Pr device to perform either secure boot with CARE or RA (as shown in steps 5 and 6). The details of system design and working are covered in subsection §IV-A and subsection §IV-B.

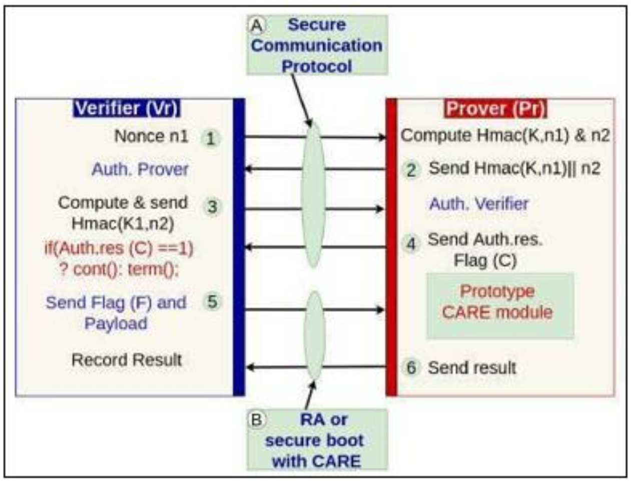

Figure 1: Highlights the proposed *SRACARE* system design flow. It represents the lightweight authenticated secure communication protocol and a new RA and secure boot architecture using custom *CARE* module

2.2 Related Work

Previous work presented in [31], performs system-level verification by writing specification and code in dafny [32] language, which supports automated verification using Z3 [33] SMT solver. Their tools convert dafny code to boogieX86 [34] verifiable assembly language. The entire system is verified at assembly level using boogie verifier [35]. The work presented in [36] formally verifies the UEFI secure boot system by validating PCR's content using TPM. Another co-verification approach shown by [37] uses instruction-level abstraction (ISA) of hardware and applies SMACK solver to formally verify specific security properties of access control and DMA. Recent work in [12] demonstrates a framework for adversary modeling and security specification problem, followed by verification by using hyperfuzing. Another recent implementation [38] uses security properties specific hardware abstraction in software and uses SMACK solver. Vrased [16] verifies the hardware module using Linear Temporal Logic (LTL) specifications and forces the system to reset upon security properties failure.

Therefore, previous research work for end-to-end security co-verification can be divided into two categories: 1) information flow analysis [39], [40] and 2) property verification through model checking [16], [38], [41]. FVCARE belongs to the second category as it uses a software model checker to verify the security properties of complex Hw-Fw interactions. The general techniques of hardware abstraction into software and using software model checkers on the composition to verify Hw-Fw interactions are not new [16], [38], [42]. However, the end-to-end co-verification of security properties and specification (as per subsection IV-C) for SRACARE based systems are yet to be explored.

For example, in Fig 1 the Pr device computes and checks the digest of each flash frame during the secure boot process. If the verification fails, the RE re-flashes the correct flash memory region, locks the write access, and continues the subsequent boot process. In this case, an attacker can change the recovery code's start location or redirect the system to measure boot integrity from the wrong memory region. The device requires adequate Physical Memory Protection (PMP) to prevent the write access to configuration registers and redirection of the code execution. Such scenarios require verification of hardware firmware and interaction, and any error can result in a security failure. FVCARE focuses on concrete multi-level model checking (not just bounded) experiments along with showcasing automated hardware verification techniques, which distinguishes it from previous works.

3. Verification Framework Design Challenges

The scalable hardware firmware co-verification framework design faces three major challenges: 1) correct system-level abstraction, 2) definition of security properties, and 3) co-verification technique implementation.

3.1 System Design Abstraction

The system to be verified can be represented either hardware abstraction as software or firmware/software mod- ules can be represented in hardware-based models. Both techniques require precisely captured sequential states of the hardware, firmware, and interacting modules. The in- correct model representation can lead to invalid verification results, a badly designed and attack-prone system. Therefore, precisely defined system security properties and boundaries for each hardware firmware components functioning are critically important for design abstraction. Section V-A covers the available types of abstraction models and the approach used by FVCARE.

3.2 Security Properties Specification

Another challenge is a system and security property specification. The hardware/firmware-based registers are setting, and component initialization, Atomicity, based temporal logic of the system can be verified by Computational Tree Logic (CTL) or Linear Temporal Logic (LTL). However, temporal logic cannot represent security properties such as controlled invocation, confidentiality, and availability. They can be verified by information flow properties analysis. Furthermore, specification of the shared system interconnect (bus) and specific SPI boundaries specification requires LTL, assertions, and weakness detection to protect the devices from [43] attacks. A combination of system security specification tools is required for system representation and exhaustive analysis.

3.3 System Verification Techniques

The co-verification of the security properties specified in either LTL, assertions, or another language needs to be checked for correctness and security assurance. This checking can be performed using Theorem Proving (TP) or Model Checking (MC). The Satisfiability Modulo Theories (SMT) solvers can be used for theorem proving. The model-checking can be used to verify the system correctness properties of finite-state transition systems [44], [45]. The model checking can be further classified into two types: 1) Unbounded model checking explores all reachable system transition states. 2) Bounded Model Checking (BMC) [46] restricts the search to all states reachable within the first k (bound) transitions of the system. Therefore, the selection of proper techniques becomes a crucial design component to perform exhaustive system verification. Following sub-section V-A covers the methodology used by FVCARE.

4. Overview of SRACARE

Before going into the details of verification methodology, this section covers the summary of system design and operation of SRACARE [30] based system used for formal verification in proposed FVCARE.

a) System Design

SRACARE system's top-level design overview is presented in Fig 2. The core security enhancing features of SRACARE based system are: 1) It implements lightweight, secure communication protocol and 2) It demonstrates the lightweight implementation of secure boot system with on board recovery engine (by using *CARE* module). It also implements sample RA architecture for run-time software state assurance of the Pr device. The notations and definitions used for the communication are listed in Table I. The detailed

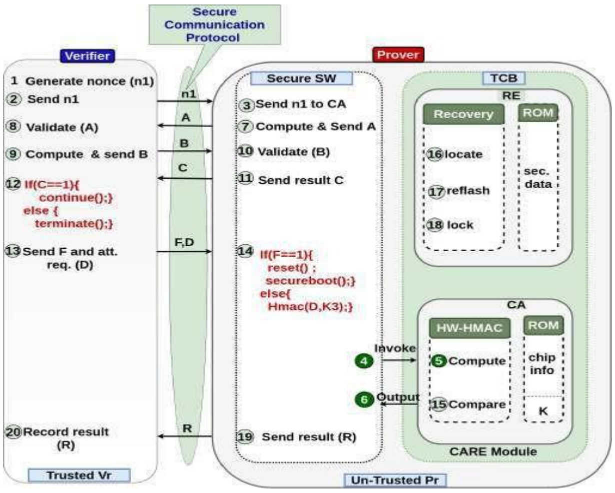

Figure 2: Highlights the system design and key contributions of *SRACARE*: 1) Novel lightweight, secure authenticated communication protocol (steps 1 to 12), and 2) Secure boot with *CARE* and remote attestation architecture for the Pr device (steps 13 to 20)).

working of the secure communication protocol (steps (1) to (12) from Fig 2) is covered in subsection §IV-B. 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 [3], [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 A3 security properties listed in section §IV-C. Fig 3 shows the internal architecture design of Pr device to satisfy the security properties from A1, A2, A4 to A12 from subsection §IV-C. *SRACARE* based P_r system follows design choices ❶ to ❺, 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 system operation of *SRACARE* based 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 secure communication starts when the Vr sends nonce n1 to the Pr device.

Table 1: Notations and description

Notation	Description
n1	Vr's nonce for freshness
n2	Pr's nonce for freshness
	n2 = Hmac(K, T)
	T = hash(CHIP INFO.) ⊕ n1
K	Symmetric key for HMAC
Hmac(K, m)	$H((K' \oplus 0x5C5C) \Vert H((K' \oplus 0x3636) \Vert m))$
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
Saddr	Start address of flash memory for hashing
L	Lenth of the memory region to be hashed
R	Final Result
K'	*H(K) K is larger than the block size K otherwise*
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	K1= (Hmac(K, n1) ⊕ n1 ⊕ n2))
$\Vert$	Denotes concatenation
⊕	Denotes bitwise exclusive or (XOR)
CA	Code Authentication
RE	Resilience Engine
RA	Remote Attestation

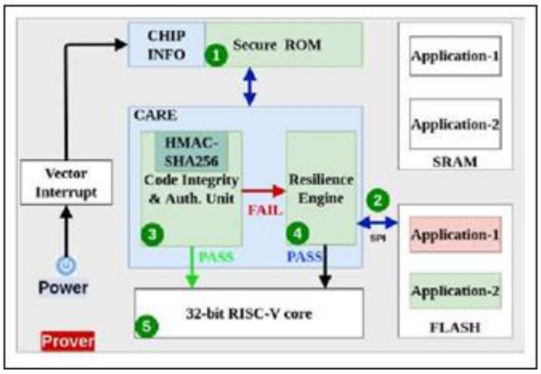

Figure 3: Shows the architecture design of *SRACARE* based P_r system, highlighted are the key design modules. The pass arrows indicate that only the known good code will be allowed to be executed on the RISC-V core at any given time.

The un-trusted Pr device uses novel n2 generation techniques by computing Hmac(K, T).

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

Where T is computed by xoring the digest of first 16 Bytes of the chip info memory and n1. The term CI_{start} indicates the starting location of Chip Info (CI) memory. 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, 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

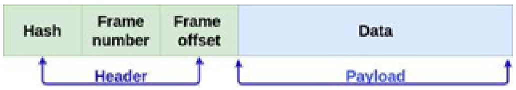

Figure 4: Represents the frame data structure. The header contains the digest of the entire frame, frame number, and flash offset location. The payload contains corresponding data for each frame.

Section of the data frame contains the digest of the 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 the Hash- based Message Authentication Code's (HMAC) feature for signing (authenticating) the data and the 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, reflashes the corrupted flash memory region with the known good software code from secure ROM, and locks 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).

c) Security Properties

FVCARE has identified twelve (A1- A12) security properties for targeted SRACARE [30] based system with secure boot, RA, and onboard recovery needs to satisfy for end-to-end co-verification. They can be broadly classified into four domains: 1) System Initialization & Secure Communication, 2)Key Protection, 3)Safe Execution, and 4) Safe Recovery.

1) System Initialization & Secure Communication: This subsection defines required security properties during the startup of the system & peripherals initialization. It also focuses on defining security properties for derived keys generation and device authentication at the Pr side. **A1. Start-up Checking:** The startup security properties require the correct implementation of start addresses and range of ROM, RAM, flash memory regions, and MMIO device mapping, based on the system's design specification. **A2. Peripheral Initialization:** This security property includes initialization of system registers, flash controller, SPI, and baud rate setting of UART. It involves the function calls from firmware to initialize the respective hardware modules (UART, SPI) and registers. **A3. Secure Communication:** The design under test uses a secure device authentication protocol for Pr & Vr devices authentication. As discussed in subsection IV-B, it uses novel derived key and nonce generation techniques. This security property requires proper derived key and nonce generation techniques.

2) Key Protection: This property ensures that all the secure device information, crypto, and derived keys are stored in secure ROM regions and protected by access control policies. **A4. Key Confidentiality:** This security property validates that the secure key (K) and derived K' are stored in a protected ROM memory region. **A5. Access Control Enforcement:** It defines the PMP access control policies to protect the system from unauthorized memory accesses. It involves both hardware and software system modules.

3) Safe Execution: This property ensures the correct implementation, controlled invocation, and un-interrupted execution of the secure boot code. **A6. Functional Correctness:** This security property requires the implementation and functional correctness of hardware-based crypto-core. **A7. Atomicity:** This specification ensures that once triggered; the code execution should not be interrupted. **A8. Error Free Execution:** All the hardware (IPs) and software sub-modules should have error-free execution. **A9. Controlled Invocation:** The security property defines no interrupt execution, DMA operations, and debugger usage are allowed during the secure boot process.

4) Safe Recovery: The correct implementation and error-free execution of RE sub-module code execution for recovery. of **A10. Attack Detection:** This security property ensures that the corrupted flash memory region is identified correctly during the secure boot process. **A11. Secure Reflash:** This property defines ensures that the affected device is re-flashed with the appropriate recovery code (from secure ROM). It requires validation of the start address and size of the flash memory and recovery data. **A12. Access Controls:** This security property ensures that proper access control policies are applied after recovery code reflash from secure ROM. It protects the device from future flash modification attacks.

d) Scope of Verification

This work provides a formal co-verification framework for SRACARE based system with secure boot, RA, and on-board recovery. All system specification, security properties, and the hardware-software modules setting & interactions are formally verified. However, formal verification of the processor is out of scope for this work. The hardware system model (crypto-core and other TCB modules) are represented in Register Transfer Level (RTL), and software - boot process, bootstrapping, and resilience engine codes are written in C programming language. The functionality and security properties are specified in Linear Temporal Logic (LTL) and assertions. The model checker NuSMV [18] is used for hardware verification. The RTL to SMV model specifications are generated using Verilog2SMV [47] tool. For the formal software verification, Frama-C [17] tool with three different plugins was used for functional, specification, weakness prediction, and LTL model checking.

5. Formal Verification Approach

This section covers the FVCARE's approach for end-to- end system modeling, abstraction, and formal verification.

a) Formal Co-Verification Methodology

As discussed earlier, the end-to-end security properties verification of a system with secure boot, RA, and recovery engine becomes a complex problem, and we argue that only hardware verification or verification of only software abstraction of hardware can miss out on critical security bugs during the security properties setting or interaction. Therefore, the FVCARE framework proposed two-step end-to-end security properties verification for the system under test. In the first step of the verification, 1) it verifies the security-critical hardware components of the FVCARE system using hardware verification technique and 2) for the overall system and software verification, it uses source level (C) abstraction of hardware technique.

A pictorial overview of the verification methodology is shown in Fig 5. The goal is to co-verify the firmware, software, and interacting hardware components. Fig 5(a) shows firmware verification is a complex task as it involves multiple interactions between software and hardware modules. In the first step, Boot_Rom firmware code (FW-1) initializes the UART, internal registers with boot settings (HW module 1), and transfers the control to second stage Boot_Strap firmware code (FW-2). The FW-2 code initializes the SPI, applies the access control policies (HW module 2), and performs a chain of integrity & authenticity measurements using hardware crypto-core (HW module 3). The hardware verification method is shown in Fig5(b) and covered in subsection V-B. The framework uses the source level abstraction of the hardware module for end-to-end security properties verification by software abstraction approach. Software verification is presented by Fig 5(c) and covered in subsection V-C.

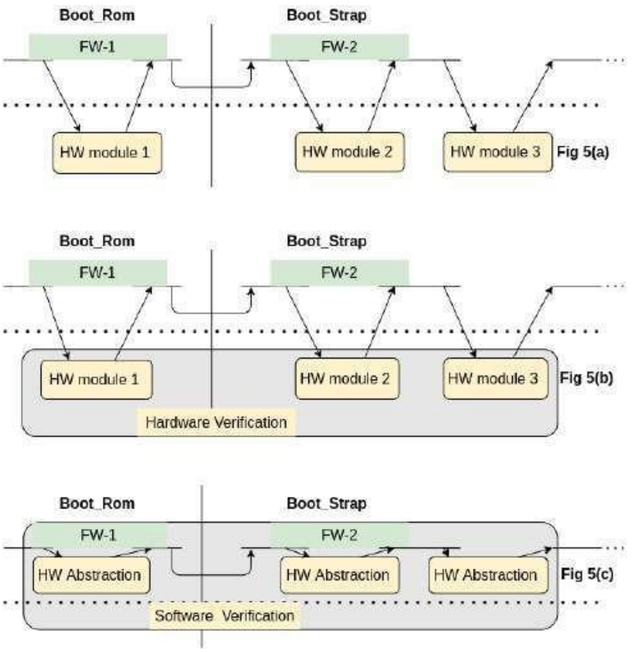

Figure 5: Formal Verification Methodology

b) Hardware Verification Technique

The framework formally verifies the security-critical hardware component - crypto-engine HMAC-SHA256 's security properties and functional correctness. It also verifies the access control policies and internal registers settings for UART's baud rate, SPI's initialization, and the flash controller. The formal verification of all other hardware modules, including the processor, is out of this work scope.

The hardware module listed above are written in system Verilog. Linear Temporal Logic (LTL) is used to formalize the system specifications, security properties (A1-A6) from subsection IV-C, and invariant that should hold throughout boot code execution. FVCARE automates the system Verilog to SMV conversion process by using Verilog2SMV [47] tool. It then uses the NuSMV [18] model checker to verify the correctness of LTL specifications. Upon failure of the formal verification, the hardware module was redesigned for security assurance.

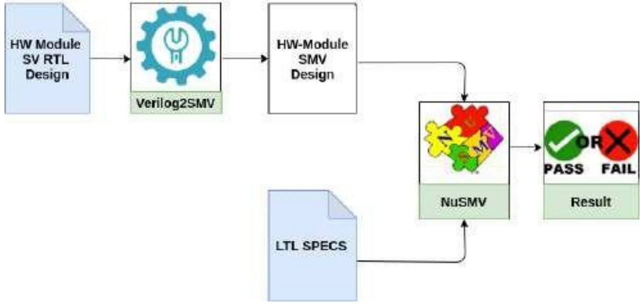

Figure 6: Top-level design of hardware verification framework. The RTL design is converted to SMV using verilog2SMV tool. The hardware specifications and security properties represented in LTL logic are verified using NuSMV (SMT solver) and pass or fail results are generated.

c) Software Verification Technique

For end-to-end system modeling Instruction Level Abstraction (ILA) of the software of the system. ILA approaches [9]–[11] involves the deep understanding and cycle-accurate conversion of the software system into instructions for that processor, and it cannot scale for fast pace changing markets. Another approach uses the hardware mod- ules' abstraction into suitable software code (C programs), as shown in Fig 5(c). After source-level abstraction, FVCARE uses Frama-C [17] (FRAmework for Modular Analysis of C code) for software verification tool.

The Frama-c tool can be used for buffer-overflow, pointer safety, exceptions, termination, K-induction, and invariant checking. It can also be used for specific system properties checking using assertions and LTL specifications. The Frama-c framework has a collection of interoperable, scalable, and sound software analysis tools. In this work, the Frama-c framework is used with three main plugins: 1) Abstract interpolation-based Value plugin 2) Weak prediction (WP) for deductive verification, and 3) System specification & security properties verification using LTL specifications.

- The abstract interpretation framework based on the VALUE plugin is used to compute the over-approximations of possible values of program variables at each program end-point. It uses formal behavioral specification language ACSL (ANSI/ISO C Specification Language) to specify the C program's functional properties and contracts. ACSL uses clauses for precondition verification, ensures clauses for post-condition verification, assign for global variables, and loop invariant clauses are used for loop iterations.
- The Weak Prediction (WP) plugin verifies that the given C code satisfies its specification expressed as ACSL annotations. The weak prediction provided by [48] reduces any deductive verification problem to establishing the validity of first-order formulas called

verification conditions. FVCARE framework then uses Alt-Ergo SMT solver to prove the verification conditions generated by WP.

- The Aorai plugin is used to verify the system properties represented in LTL specifications. The software verification framework setup with Frama-c and Aorai plugin is shown in Fig 7.

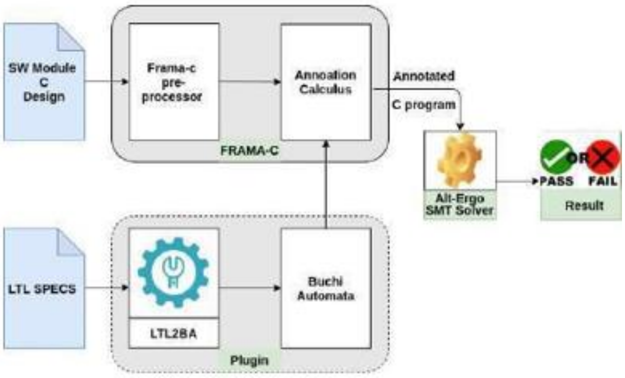

Figure 7: Presents the architecture design of the proposed framework. The pass arrows indicate that only the known good code will be passed to the RISC-V processor core for execution in any given case.

The Frama-c pre-processor module converts the C program into annotation calculus. The LTL2BA tool converts the LTL specification into Buchi automaton, combined with annotation calculus to generate an annotated C program. Alt-Ergo SMT solver is used to prove the verification conditions with fail or pass results formally. The functions and code were modified to satisfy the security properties specified in A1 to A12.

6. Evaluation

This section covers details of verification techniques selection for each of the security components in SRACARE based design under test. It shows verification results and timing analysis. Finally, it compares the proposed FVCARE framework with state-of-the-art secure boot, RA, and formally verified systems. Based on the security properties specification in subsection IV-C and system operation in subsection IV-B, the end-to-end system verification is divided into the following subtasks.

a) Verification of System Initialization

The formal verification of start addresses and range of ROM, RAM, flash memory regions, MMIO device mapping (security property A1), and registers initialization (A2) is performed by using the hardware verification technique V-B. The peripheral initialization for security property A2 is verified using co-verification techniques.

b) Verification of Secure Communication

To limit the secure formal communication verification problem, the FVCARE focuses only on verifying the Pr side security properties. Therefore, this work's verification efforts assume that nonce n1 and Flag selection values are provided to the prover (Pr) device. The LTL model was designed based on the system specifications to validate novel nonce generation techniques, and assertions were passed.

c) Verification of Secure Boot

The secure boot verification process involves multiple transactions between Hw-fw. Therefore, it uses software abstraction of the hardware. The verification of security properties A3 to A9 was also included in this work. The security properties were passed in LTL formulas, assertions, and annotations in Frama-C. The hardware verification of crypto- core (HMAC-SHA256) is covered in subsection VI-F.

d) Verification of Remote Attestation

To reduce the complexity of verifying RA communication protocol, FVCARE checks the preset flag condition (F=1 from Fig 2), start location, and the flash region size. The annotations validation, LTL specification checking for digest computation is performed using Frama-C. Formal verification of 1 KB of the digest computation requires 0.02s on intel NUC i-7 8th generation running @ 3.4GHz.

e) Verification of Resilience Engine

Since the RE module was implemented in software, the verification is also performed using the software-based Frama-C tool with plugins. The important verification parts in this are to validate frame numbers and flash memory locations. The framework also verifies proper access locks during and after the code reflash.

f) Verification of Security properties

The security properties specification, formal verification approach (Hw or Fw based), execution time, and results are represented in Table III. It also provides details about hardware or firmware-based formal verification techniques usage for set property. Selected set of the security properties, their verification technique (Hw/Fw), the time required on Intel Next Unit of Computing (NUC) i7 @3.4Ghz.

Table III: Security Properties verification results on i7-NUC @3.4GHz

Security Properties Specification	*Time (s)*	Hw	Fw	Results
Start-up Checking	0.02	yes	no	✓
Peripheral Initialization	0.03	yes	yes	✓
Key Confidentiality	0.02	yes	no	✓
Access Control Enforcement	0.03	yes	yes	✓
Controlled Invocation	0.02	yes	no	✓
Attack Detection	0.02	yes	yes	✓
Correct Frame Locations	0.02	no	yes	✓
Validate Frame Size	0.02	no	yes	✓
Functional Correctness	0.2	yes	yes	✓

Note that FVCARE verifies the crypto-core's functional correctness using hardware verification as discussed in subsection V-B. Furthermore, the system's functional correctness of secure boot and RA features is validated using the hardware approach's software abstraction. The formal verification of the secure boot for the test application of 5.6 KB takes 0.2 seconds.

7. Comparison with the state-of-the-art solutions

For the state-of-the-art comparison, FVCARE has identified several recent implementations of secure boot and RA techniques as listed in Table II. As can be seen from Tabel II majority of the available, secure boot and RA implementations focus on detecting and preventing malicious code modification attacks. However, it lacks protection from attacks and mostly restarts the system or leaves it in a non-operational state. Recent implementations Healed and [49] provides recovery, but they do not have secure boot, RA, and formal verification support. Other implementation [16] shows formal verification of RA module using LTL specification, with two unsuitable design choices: 1) it uses software-based crypto-core (HACL*) for digest computation in RA, 2) it does not have secure boot support, and 3) it recommends systems reset to prevent the attacks. Work presented by [38] uses the instruction-level abstraction of hardware approach for formal verification of security primitives such as secure boot. ILA approach is very restrictive, ISA specific, and limited to scale. Therefore, not suitable for scalable end-to-end system verification. Another work presented in [41] demonstrates the use of the source-level abstraction of the hardware and bounded model checking for industry-standard SoC's security verification. Furthermore, Recent implementations CARE [29] and SRACARE [30] demonstrates resilient small embedded system design with secure boot, RA and on-board recovery techniques. For various security reasons and practical use-cases, our hypothesis required secure boot, RA, and onboard recovery such as CARE [29], and SRACARE [30]. FVCARE uses the source- level abstraction of the hardware approach and enhances the model checking capabilities by using the Frama-C tool with different K-induction plugins, Weakness prediction, assertion, and bounded model checking using LTL. Thus, FVCARE is the first implementation that integrates hardware and software verification methods and demonstrates the end-to-end co-verification technique for SRACARE based systems with secure boot, RA, and onboard recovery mechanisms.

8. Conclusion

FVCARE provides the end-to-end co-verification framework for SRACARE based systems with secure boot, RA, and onboard recovery. It uses the abstraction of hardware as a software technique for formal verification of design specifications, security properties, and the system's functional correctness. Also, it uses hardware verification techniques for verification of initial system and registers settings, access control policies. It demonstrates formal verification of hardware using Linear Temporal Logic (LTL) properties and using model checking NuSMV tool. FVCARE leverages the software abstraction of the hardware approach for software verification and uses a novel Frama-C framework with different plugins for formal verification of the system rep- resented as software abstraction of the hardware. FVCARE demonstrates the first practical implementation of a formal co-verification framework.

Table II: Qualitative comparison between secure boot/RA techniques targeting lightweight embedded devices

Parameters	[30], [29]	Healed	Ref. [49]	Ref. [25]	Sanctum	Ref. [38]	Ref. [16]	Ref. [41]
Design Type	Hybrid	SW	Hybrid	HW	Hybrid	Hybrid	Hybrid	Hybrid
Secure Communication	yes	no	no	yes	yes	yes	no	no
Secure boot	yes	no	no	yes	yes	yes	no	no
Remote Attestation	yes	no	no	no	yes	no	yes	no
Malicious Code Modification Attacks Detection	yes	yes	yes	yes	yes	yes	yes	yes
Malicious Code Modification Attacks Protection	yes	no	yes	yes	yes	yes	yes	yes
Recovery from Malicious Code Modification Attacks	yes	yes	yes	partial	no	no	no	no
Formal Verification of Hardware	no	no	no	no	no	yes	yes	yes
Formal Verification of Software	no	no	no	no	no	yes	yes	yes
Formal hardware/software co-Verification	no	no	no	no	no	yes	yes	yes

References

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

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

[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] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang, "Challenges and trends in modern soc design verification," *IEEE Design Test*, vol. 34, no. 5, pp. 7–22, 2017.

[8] S. Ray and Y. Jin, "Security policy enforcement in modern soc designs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '15. IEEE Press, 2015, p. 345–350.

[9] Schmidt, C. Villarraga, J. Bormann, D. Stoffel, M. Wedler, and W. Kunz, "A computational model for sat-based verification of hardware-dependent low-level embedded system software," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 711–716.

[10] Grobe, U. Ku¨hne, and R. Drechsler, "Hw/sw co-verification of embedded systems using bounded model checking," in *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 43–48. [Online]. Available: https://doi.org/10.1145/1127908.1127920

[11] S. Malik and P. Subramanyan, "Invited: Specification and mod- eling for systems-on-chip security verification," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[12] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for soc security validation," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[13] S. Ray, N. Ghosh, R. J. Masti, A. Kanuparthi, and J. M. Fung, "Formal verification of security critical hardware-firmware interactions in commercial socs," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3323478

[14] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co- designs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/ 10.1145/3061639.3062253

[15] B. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in socs using instruction- level abstraction for hardware*," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[16] 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

[17] F. Bobot, "Verilog2smv: A tool for word-level verification," https: //git.frama-c.com/pub/frama-c, 2016.

[18] Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, ser. CAV '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 359–364.

[19] "TPM wiki," https://en.wikipedia.org/wiki/Trusted Platform Module, 2010.

[20] 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

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

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

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

[25] 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.

[26] NSA Cyber Report, "UEFI DEFENSIVE PRACTICES

[27] GUIDANCE," https://www.nsa.gov/Portals/70/documents/what-we-do/cybersecurity/professional-resources/ctr-uefi-defensive-practices- guidance.pdf, 2017.

[28] 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.

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

[30] Dave, N. Banerjee, and C. Patel, "Care: Lightweight attack resilient secure boot architecture with onboard recovery for risc-v based soc," https://arxiv.org/pdf/2101.06300.pdf, 2020.

[31] ——, "Sracare: Secure remote attestation with code authentication and resilience engine," in *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2020, pp. 1–8.

[32] Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated full-system verification," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 165–

[33] 181. [Online]. Available: https://www.usenix.org/conference/osdi14/ technical-sessions/presentation/hawblitzel

[34] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[35] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[36] J. Yang and C. Hawblitzel, "Safe to the last instruction: Automated verification of a type-safe operating system," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 99–110. [Online]. Available: https://doi.org/10.1145/1806596.1806610

[37] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier

for object-oriented programs," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[38] M. Nelson, "Modeling the secure boot protocol using actor net- work theory," https://scholarspace.manoa.hawaii.edu/bitstream/10125 / 62288/2017-12-ms-nelson.pdf, 2017.

[39] S. Ray, N. Ghosh, R. J. Masti, A. Kanuparthi, and J. M. Fung, "Formal verification of security critical hardware-firmware interactions in commercial socs," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3323478

[40] Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik, "Formal security verification of concurrent firmware in socs using instruction- level abstraction for hardware*," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[41] M. Balliu, M. Dam, and R. Guanciale, "Automating information flow analysis of low level code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1080–1091. [Online]. Available: https://doi.org/10.1145/2660267.2660322

[42] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. M. Fung, "Verifying information flow properties of firmware using symbolic execution," *2016 Design, Automation & Test in Europe Conference & Exhibition DATE*, pp. 337–342, 2016.

[43] S. Ray, N. Ghosh, R. J. Masti, A. Kanuparthi, and J. M. Fung, "Formal verification of security critical hardware-firmware interactions in commercial socs," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3323478

[44] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co- designs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/ 10.1145/3061639.3062253

[45] 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.

[46] K. L. Mcmillan, "Symbolic model checking," https://apps.dtic.mil/sti/ pdfs/ADA250924.pdf, 1993.

[47] M. C. O. Grumberg and D. Peled, "Model checking," MIT Press, 1999.

[48] Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," ser. Advances in Computers. Elsevier, 2003, vol. 58, pp. 117 – 148. [Online]. Available: http://www. sciencedirect.com/science/article/pii/S0065245803580032

[49] A. C. A. G. M. Roveri and R. Sebastiani, "Verilog2smv: A tool for word-level verification," http://disi.unitn.it/rseba/papers/ verilog2smv-proceeding-version.pdf, 2016.

[50] W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," Commun. ACM, vol. 18, no. 8, p. 453–457, Aug. 1975. [Online]. Available: https://doi.org/10.1145/360933.360975

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

Author Profile

Avani Dave PhD. Candidate, CSEE, University of Maryland Baltimore County, MD, USA.

Nilanjan Banerjee, Professor, CSEE, University of Maryland Baltimore County, MD, USA.

Chintan Patel *Member, IEEE,* Associate Professor, CSEE, University of Maryland Baltimore County, MD, USA.