A Novel Approach to Secure Execution in Embedded Systems: Custom ISA and Compiler Binding

mariaayub

I. Abstract

II. Introduction

III. Related Work

Embedded platforms are increasingly deployed in environments where adversaries can obtain physical access, interact with programming and debug interfaces, and attempt firmware replacement. Prior work addresses these threats along several directions: (i) BootROM-centered secure boot, (ii) securing longer boot chains (e.g., OS/FPGA-style stacks), (iii) debug and test interface security, (iv) hardware-enforced trusted execution and minimal TCB designs, (v) software-only hardening, and (vi) compiler, toolchain, and binary-translation infrastructures that make low-level enforcement mechanisms practical to deploy. This section reviews these approaches and motivates instruction-triggered, key-gated authorization with toolchain support.

A. BootROM-Based Secure Boot and Early-Stage Integrity

Secure boot typically anchors trust in immutable first-stage code (BootROM/ROM-resident loader) that authenticates subsequent software before execution. Bin *et al.* design BootROM support for secure boot mode, reinforcing the BootROM's role as the earliest enforcement point and focusing on validating system images during startup to prevent unauthorized firmware execution [1]. Complementing this, Rashmi R. V. and Karthikeyan survey secure boot techniques for embedded applications, highlighting common architectural patterns and practical constraints (e.g., resource limits, deployment workflow) that shape how image authentication is implemented [2]. While these works establish boot-time authenticity as a baseline, they leave open how *post-boot* privileged actions (e.g., enabling debug/program access) should be authorized once some code is running—particularly under physical-access threat models where adversaries can iteratively probe and reflash.

A related line of work emphasizes extending boot trust into device identity and measurement chains. The DICE architecture specifies a mechanism for device identity composition and chained measurement, which is widely used as a conceptual basis for building scalable roots of trust and attestation for constrained devices [3].

B. Boot-Chain Protection for Richer Stacks (Embedded Linux / FPGA Systems)

As embedded systems incorporate more complex boot sequences (multiple software stages, configuration artifacts, and OS components), integrity must extend beyond a single image check. Devic *et al.* explicitly address this problem for an embedded Linux boot flow on FPGA, securing the boot process against tampering and replay attacks by verifying artifacts in a multi-stage chain [4]. This class of work motivates defenses that remain robust when attackers target whichever stage is easiest to manipulate.

C. Debug/Test/Programming Interface Security

Debug and test access are widely recognized as high-impact attack surfaces because they provide invasive control over execution, memory, and device state. IEEE 1149.1 standardizes the test access port and boundary-scan architecture that underpins much of the modern debug/test ecosystem [5]. Beyond standardization, boundary-scan/test-mode pathways can enable attacks when an adversary can enter privileged test states [6]. Recent work also explores adding explicit authentication mechanisms to debug access; for example, Lapeyre *et al.* propose a lightweight JTAG authentication IP intended to support secure device testing while restricting unauthorized debug usage [7].

D. Hardware-Enforced Trusted Execution and Minimal-TCB Approaches

TrustZone popularized a hardware partitioning model that enables a "secure world" isolated from the normal execution environment [8]. Open-platform efforts such as Keystone demonstrate how trusted execution environments (TEEs) can be engineered with explicit, auditable interfaces [9]. In parallel, systems such as TrustVisor, Flicker, and Memoir motivate minimizing trusted code while maintaining measured execution and continuity guarantees [10]–[12]. For embedded-scale devices, TrustLite proposes isolation mechanisms tailored to constrained systems [13].

E. System Assurance via Formal Verification

Because low-level enforcement mechanisms are security-critical, prior work emphasizes formal verification as a path to high assurance. The seL4 microkernel demonstrates formal verification at the operating-system kernel level [14], [15].

F. Software-Only Hardening: Obfuscation and Key-Dependent Checks

Software protection techniques deter tampering and reverse engineering by transforming code or embedding secret-dependent checks. Collberg *et al.* provide a foundational taxonomy of obfuscating transformations [16].

G. Toolchain Infrastructure, ISA Extensibility, and Deployable Low-Level Mechanisms

Many enforcement mechanisms require reliable support from the compiler and binary utilities. Shen *et al.* present LLBT, an LLVM-based static binary translator [17]. Zakai introduces Emscripten, an LLVM-to-JavaScript compiler [18]. Finally, RISC-V's explicit support for extensibility motivates instruction-level mechanisms that remain compatible with conventional toolchains and disassembly workflows [19].

IV. System Architecture

A. Architectural Components

Processing core. The target is a standard embedded CPU extended with decode support for a security instruction. The extension is minimal: it adds an opcode pattern for the authorization instruction and a control path to invoke the security subsystem without exposing secret material on the normal memory hierarchy.

Security-instruction interface. The decode/execute path recognizes a custom instruction (`kc`) and forwards its operands and execution context to the security subsystem. The forwarded metadata includes the immediate field (used to convey an authorization token fragment) and, optionally, privilege/boot-stage context to ensure the authorization sequence is invoked at the intended point (e.g., BootROM/early boot).

Protected key storage (NVM). The reference authorization material is stored in a protected non-volatile region that is *not memory-mapped* into the CPU's standard load/store address space. Access to this region is mediated only through the verification datapath, preventing untrusted firmware from reading the key through conventional bus transactions.

Compare/verification unit. A constant-time comparator internally reads the reference value and compares it against a candidate token derived from the `kc` operands and context. The unit outputs only a one-bit decision `match` $\in \{0,1\}$ (and optional status), and never exposes the reference value externally.

Authenticator (control FSM). A security control finite-state machine (FSM) consumes the verification outcome and drives two gating outputs: (i) `exec_enable`, which releases execution into normal firmware, and (ii) `debug_enable`, which unlocks invasive debug/program access.

Debug-unit gate. The debug subsystem is disabled by default. It is enabled only when `debug_enable` $= 1$, ensuring that invasive debug operations are unavailable prior to successful authorization.

Host binding database and audit log. A centralized host-side database stores the binding relationship between *host identity*, *device identity*, and *authorization material*. In addition to the current binding record, the database maintains audit data (e.g., latest programming event, timestamps, toolchain version, and device identifier) to support traceability and accountability.

B. Key Material and Binding Model

Let $K \in \{0,1\}^{26}$ denote a 26-bit authorization key (or verifier) provisioned into protected storage. The key is derived from both *host-specific* and *device-specific* information to implement binding that is resistant to trivial reuse across endpoints.

Host-side generation and binding. A host authorization key is generated from a mixture of host identifiers and dynamic inputs, and is bound to both the authorized host record and the target device record. Concretely, the input material includes: (i) `timestamp`, (ii) `machineID`, (iii) `hostUUID`, (iv) `deviceUUID`, and (v) a random nonce r. Abstractly, key derivation is modeled as:

$$K = \mathrm{Trunc}_{26}\Big(\mathrm{Mix}(\texttt{timestamp}, \texttt{machineID}, \texttt{hostUUID}, \texttt{deviceUU} \tag{1}$$

where r is a nonce and $\mathrm{Mix}(\cdot)$ is a mixing/KDF function selected to meet security requirements. The database stores the binding tuple (host, device, key metadata) and maintains the latest programming log entry for auditability.

Provisioning. The device stores K (or a derived verifier reference) in protected NVM during manufacturing or authorized enrollment. Device-specific identifiers (e.g., `deviceUUID`) are recorded in the database and included in derivation so the resulting authorization material is bound to a specific target device.

C. ISA-Level Authorization Primitive

A custom instruction `kc` is used as the authorization trigger. **Form:** `kc rd, imm20` (I-type format). **Semantics:** executing `kc` supplies a candidate token T derived from operands and context; the verification unit compares T to K and the Authenticator updates state. The primary effect is gating of both execution and debug enablement.

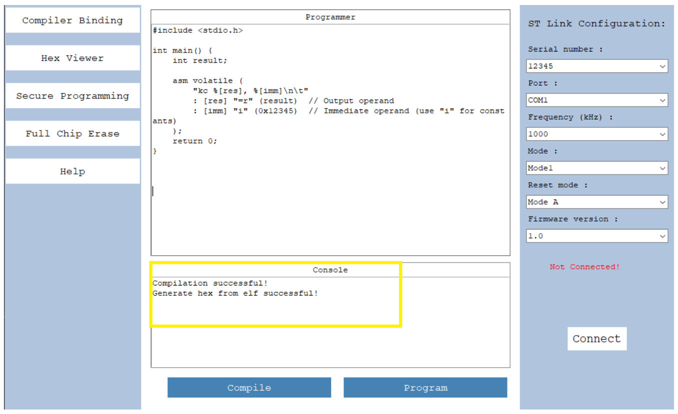
D. Authenticator FSM (Fail-Stop, Dual Gating)

We define the Authenticator FSM as: states $S = \{\textsf{LOCKED}, \textsf{VERIFY}, \textsf{AUTHORIZED}, \textsf{HALT}\}$; inputs `kc_event`, `match`; outputs `exec_enable`, `debug_enable`, `halt_cpu`. On reset, the FSM enters $\textsf{LOCKED}$ and asserts `exec_enable` $= 0$, `debug_enable` $= 0$. On `kc_event` it transitions to $\textsf{VERIFY}$. If `match` $= 1$, it transitions to $\textsf{AUTHORIZED}$ and asserts both gates; if `match` $= 0$, it transitions to $\textsf{HALT}$ (absorbing) and asserts `halt_cpu` $= 1$.

V. Results and Evaluation

A. Evidence Artifacts and Experimental Workflow

Evaluation was conducted at two levels: (i) *toolchain-level validation* confirming that the custom authorization instruction



Programmer

```
#include <stdio.h>

int main() {
    int result;

    asm volatile (
        "kc %[res], %[imm]\n\t"
        : [res] "=r" (result)  // Output operand
        : [imm] "i" (0x12345)  // Immediate operand (use "i" for const
ants)
    );
    return 0;
}
```

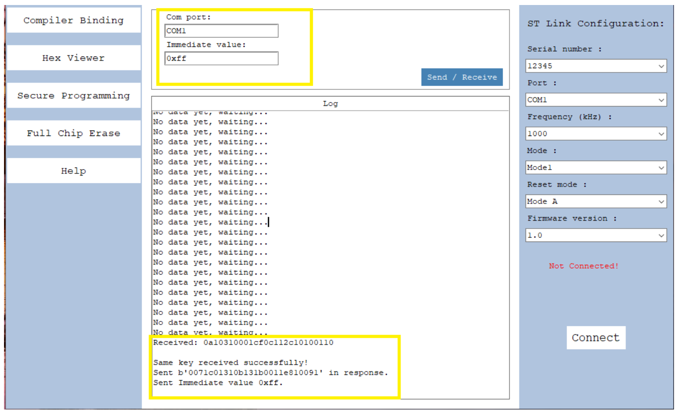
Console

```
Compilation successful!
Generate hex from elf successful!
```

ST Link Configuration:

```
Serial number :
12345
Port :
COM1
Frequency (kHz) :
1000
Mode :
Mode1
Reset mode :
Mode A
Firmware version :
1.0
Not Connected!
```

Fig. 1. Toolchain validation: successful compilation and ELF→HEX generation with a test program invoking the custom instruction.



```
Com port:
COM1
Immediate value:
0xff
```

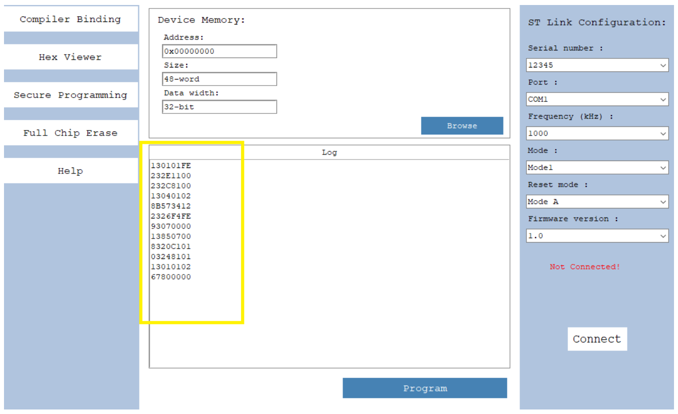
Log

```
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
Received: 0a10310001cf0c112c10100110

Same key received successfully!
Sent b'0071c01310b131b0011e810091' in response.
Sent Immediate value 0xff.
```

ST Link Configuration:

```
Serial number :
12345
Port :
COM1
Frequency (kHz) :
1000
Mode :
Mode1
Reset mode :
Mode A
Firmware version :
1.0
Not Connected!
```

Fig. 3. Authorization evidence: key exchange and match indication in the host application log.



```
Device Memory:
Address:
0x00000000
Size:
48-word
Data width:
32-bit
```

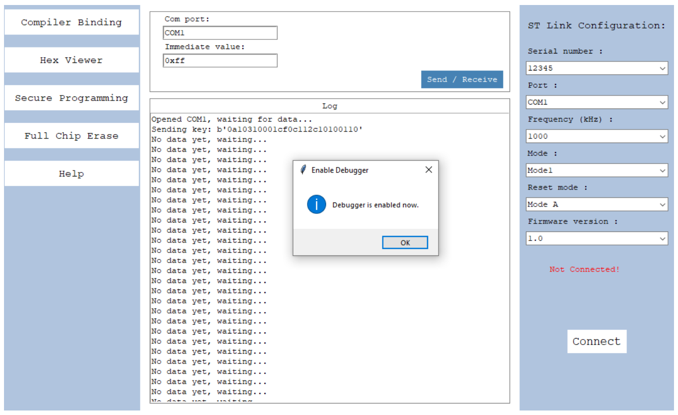
Log

```
130101FE
232E1100
232C8100
13040102
8B573412
2326F4FE
93070000
13850700
8320C101
03248101
13010102
67800000
```

ST Link Configuration:

```
Serial number :
12345
Port :
COM1
Frequency (kHz) :
1000
Mode :
Mode1
Reset mode :
Mode A
Firmware version :
1.0
Not Connected!
```

Fig. 2. Firmware inspection: hex viewer output for the compiled image.



```
Com port:
COM1
Immediate value:
0xff
```

Log

```
Opened COM1, waiting for data...
Sending key: b'0a10310001cf0c112c10100110'
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
No data yet, waiting...
```

Debugger is enabled now.

ST Link Configuration:

```
Serial number :
12345
Port :
COM1
Frequency (kHz) :
1000
Mode :
Mode1
Reset mode :
Mode A
Firmware version :
1.0
Not Connected!
```

Fig. 4. Debug gating: debugger enablement occurs only after successful key match.

is supported end-to-end (compile, assemble, link, and produce hex images), and (ii) *system-level validation* confirming that both execution and debug access are gated by authorization and that failures result in a fail-stop halt. Evidence was collected from the host-side application logs and screenshots extracted from the development artifacts.

B. Toolchain Validation

Fig. 1 shows compilation of a test program that invokes the custom instruction via inline assembly, along with successful ELF-to-HEX generation reported by the tool. This demonstrates that the custom instruction can be emitted from C code and preserved across the build pipeline without manual binary patching.

C. Firmware Inspection (HEX Viewer)

To support operator verification and troubleshooting, the host application provides a hex viewer for generated images. Fig. 2 shows an example view of the generated hex contents.

D. Authorization and Debug Enablement (Key Match)

The core functional requirement is that debug access is released only after successful authorization. Fig. 3 shows the key exchange and matching evidence in the application log, and Fig. 4 shows the resulting "Debugger is enabled" indication after a successful match.

E. Connection and Programming Workflow

Fig. 5 shows an example "Connected Successfully" state in the programming interface. Fig. 6 shows a programming session log including the `load_image` operation for a firmware ELF, demonstrating an end-to-end workflow from compilation to device programming.

F. Fail-Stop Enforcement (Halt-on-Failure)

The security policy requires halting the processor on authorization failure while keeping execution and debug disabled. Fig. 7 provides evidence consistent with a fail-stop halt state.

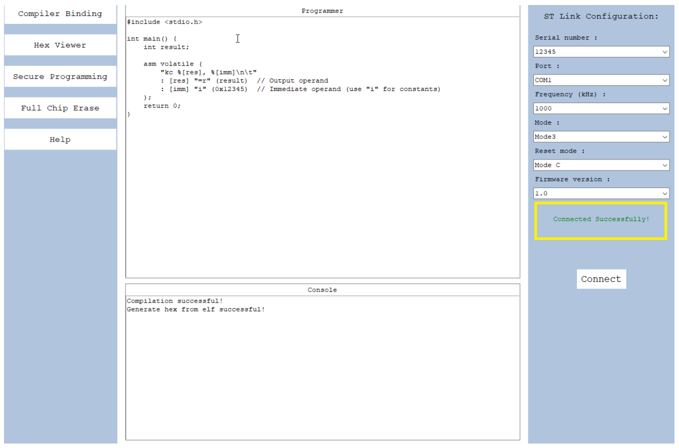
G. Summary

Table I summarizes observed behaviors supported by the collected evidence.

VI. Discussion

A. Interpretation of Results

The presented evidence supports three core properties of the proposed design: (i) the authorization primitive is usable

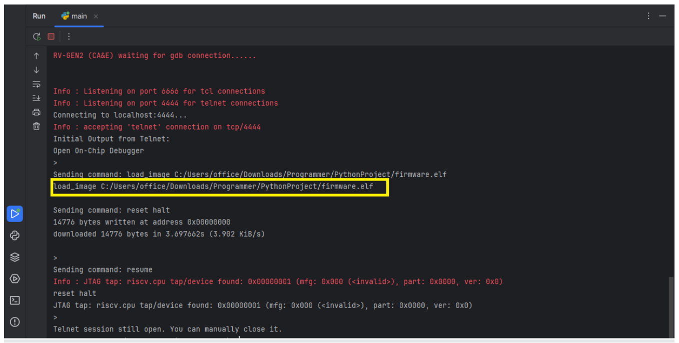


```
#include <stdio.h>

int main() {
    int result;

    asm volatile (
        "kc %[res], %[imm]\n\t"
        : [res] "=r" (result)  // Output operand
        : [imm] "i" (0x12345)  // Immediate operand (use "i" for constants)
    );
    return 0;
}
```

```
Serial number : 12345
Port : COM1
Frequency (kHz) : 1000
Mode : Mode3
Reset mode : Mode C
Firmware version : 1.0
Connected Successfully!
```

```
Compilation successful!
Generate hex from elf successful!
```

Fig. 5. Connection status: host tool indicates successful device connection.



```
RV-GEN2 (CA&E) waiting for gdb connection......


Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Connecting to localhost:4444...
Info : accepting 'telnet' connection on tcp/4444
Initial Output from Telnet:
Open On-Chip Debugger
>
Sending command: load_image C:/Users/office/Downloads/Programmer/PythonProject/firmware.elf
load_image C:/Users/office/Downloads/Programmer/PythonProject/firmware.elf

Sending command: reset halt
14776 bytes written at address 0x00000000
downloaded 14776 bytes in 3.697662s (3.902 KiB/s)


>
Sending command: resume
Info : JTAG tap: riscv.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
reset halt
JTAG tap: riscv.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
>
Telnet session still open. You can manually close it.
```

Fig. 6. Programming workflow evidence: host log includes `load_image` for the compiled firmware ELF.

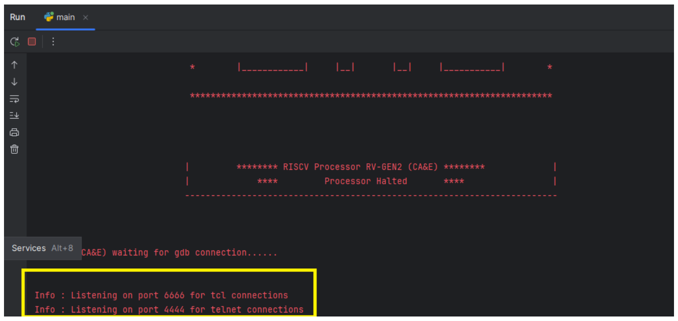
through the standard build pipeline (compile, link, and image generation), (ii) debug enablement is released only after an explicit authorization event, and (iii) the failure policy is fail-stop (halt-on-failure) with both execution and debug remaining disabled. Collectively, these properties establish that authorization is enforced *at the device* and is not solely a host-side convention.

B. Security Rationale and Expected Resistance

The architecture targets a pragmatic embedded threat surface: unauthorized firmware replacement and invasive debugging after physical access. By implementing the reference key in protected non-volatile storage that is not accessible via ordinary memory transactions, the design reduces the feasibility of key extraction by untrusted firmware through standard load/store operations. Furthermore, coupling authorization to an ISA-visible trigger integrates the policy into the execution model: authorization is invoked explicitly and can be located and audited in compiled artifacts, which is advantageous in engineering workflows where reproducibility and traceability are required.

C. Fail-Stop Enforcement (Halt-on-Failure)

Selecting halt-on-failure provides a conservative security posture. In particular, the fail-stop policy prevents the system



```
*       |___________|     |__|       |__|     |__________|        *

*******************************************************************




|        ******** RISCV Processor RV-GEN2 (CA&E) ********           |
|            ****         Processor Halted       ****               |
--------------------------------------------------------------------



CA&E) waiting for gdb connection......


Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Fig. 7. Fail-stop behavior: processor indicates a halted state consistent with halt-on-failure enforcement.

TABLE I
FUNCTIONAL VALIDATION SUMMARY FROM COLLECTED ARTIFACTS.

Requirement	Evidence
Custom instruction compiles and emits	Fig. 1
ELF→HEX generation succeeds	Fig. 1
Hex inspection supported	Fig. 2
Key exchange and match visible	Fig. 3
Debug enabled only after match	Fig. 4
Device connection workflow supported	Fig. 5
Programming log shows image load	Fig. 6
Halt-on-failure evidence	Fig. 7

from progressing into a partially-initialized state after a failed authorization attempt and reduces the opportunity for iterative online probing within a single boot session. From an engineering standpoint, fail-stop also simplifies state reasoning: the security FSM transitions either to an *authorized* mode that releases both gates, or to an absorbing halt state that releases none.

D. Trusted Computing Base Considerations

The design shifts part of the security argument from purely runtime checks to the combined behavior of hardware enforcement and the software/toolchain pipeline. Because the authorization trigger is intended to be emitted deterministically by the toolchain, the toolchain and programming workflow become security-relevant artifacts. In practice, this motivates hardening and controlling distribution of the compiler/toolchain, and recording build provenance (e.g., toolchain version and policy) as part of the programming audit trail.

E. Operationalization: Provisioning and Policy

The security guarantees depend on correct provisioning of the reference key (or derived verifier) into protected storage and consistent host-side binding policy. The operational workflow benefits from centralized records and log tracking, which support accountability and can assist incident investigation. For deployment beyond a laboratory setting, the binding policy must be complemented with explicit lifecycle procedures (enrollment, administrative recovery, and decommissioning).

VII. Future Work

A. Cryptographic Protocol Refinement

Future work should formalize the authorization exchange as a standard cryptographic protocol (e.g., nonce-based challenge–response) and bind authorization tokens to a device-unique identity. This would strengthen resistance to replay and improve interoperability with established secure-boot and attestation ecosystems.

B. Key Lifecycle Management

An operational system requires key rotation and endpoint revocation. Adding explicit rotation epochs, revocation lists, and administrative recovery procedures would enable continued security under host compromise and credential expiration.

C. Anti-Replay and Attempt Limiting Across Resets

While fail-stop constrains repeated attempts per boot session, repeated reset-based attempts remain possible in principle. Persisting attempt counters in protected state, adding exponential backoff, and requiring fresh nonces from the device can raise the cost of brute-force or replay-driven attempts.

D. Formal Methods and Assurance

Because the Authenticator FSM is security-critical, future work should apply formal verification to prove safety properties (e.g., unauthorized paths never assert `exec_enable` or `debug_enable`) and to validate that failure states are absorbing under the specified policy.

E. Overhead Characterization

The current evaluation focuses on functional correctness. A follow-on study should quantify overhead, including verification latency (impact on boot time), hardware area/power of the compare path, and any code-size impact introduced by toolchain integration.

F. Policy Generalization

The same authorization primitive can be extended to gate additional privileged operations (e.g., firmware update mode, privileged peripheral configuration, secure memory region access). This would generalize the mechanism from debug-only control to a broader platform authorization framework.

VIII. Future Work

A. Device-Bound Key Derivation and Provisioning Workflow

While the current design binds authorization to host- and device-associated identifiers (e.g., UUIDs), a complete deployment requires a well-defined provisioning workflow. Future work will formalize enrollment steps for extracting/confirming a device identifier, generating the device-bound verifier, and securely provisioning the reference value into protected non-volatile storage. This includes defining recovery procedures for legitimate re-enrollment (e.g., device replacement) while preventing unauthorized re-binding.

B. Database-Backed Policy and Audit at Scale

The host-side database can be extended from a single binding store to a policy service that manages multiple devices, multiple authorized hosts, and role-based access (e.g., operator vs. administrator). Future work will add (i) firmware metadata storage (firmware hash/version), (ii) per-device authorization history, and (iii) reproducible build metadata (toolchain version, configuration), enabling stronger provenance and easier forensic analysis.

C. Multi-Step Authorization (Key Sequencing)

The design diagrams include a staged authorization sequence (e.g., Key-1 $\rightarrow$ Key-2 $\rightarrow$ Key-3). Future work will implement and evaluate multi-step authorization in the Authenticator FSM, including strict ordering, per-step timeout constraints, and explicit failure-to-halt semantics. This can improve robustness against partial replay and makes the authorization protocol more resistant to trivial single-token reuse.

D. Hardware and Micro-Architectural Evaluation

The current results emphasize functional correctness and workflow evidence. Future work will quantify overhead and design trade-offs, including (i) authorization latency (boot-time impact), (ii) area/power cost of the compare path and control FSM, and (iii) any code-size impact introduced by toolchain integration and BootROM logic.

E. Toolchain Hardening and Regression Testing

To make the ISA extension sustainable, future work will add automated regression tests that validate encoding/decoding, compilation, disassembly, and correct placement of the authorization primitive in early boot. This includes a small test suite that checks that binaries contain the expected instruction sequences and that failure modes (halt-on-failure) are triggered under incorrect tokens.

IX. Conclusion

This work presented a hardware–software co-designed approach for securing embedded programming and execution under physical-access threat models. The proposed architecture gates *both* normal execution and invasive debug access on a device-resident authorization decision, implemented through protected key storage and a verification datapath that is not exposed via ordinary memory transactions. Authorization is invoked through an ISA-visible primitive integrated into the toolchain, improving deployability and enabling reproducible, inspectable binaries.

The evaluation demonstrated end-to-end feasibility using collected artifacts: (i) successful compilation and image generation with the custom instruction in the build flow, (ii) key exchange/matching evidence that precedes debug enablement, (iii) an operational host workflow supporting compilation, inspection, connection, and programming, and (iv) fail-stop behavior consistent with a halt-on-failure policy when authorization is not satisfied. Together, these results support the practicality of instruction-triggered, key-gated control as a foundation for restricting unauthorized reflashing and debugging in embedded systems.

References

[1] N. Bin, L. Dejian, L. Zhangjian, Y. Lixin, B. Zhihua, H. Longlong, and L. Sheng, "Research and design of bootrom supporting secure boot mode," in *2020 International Symposium on Computer Engineering and Intelligent Communications (ISCEIC)*. IEEE, 2020, pp. 5–8.

[2] R. R. V. and K. A., "Secure boot of embedded applications - a review," in *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. IEEE, 2018, pp. 291–298.

[3] Trusted Computing Group (TCG), "Device identifier composition engine (dice) architecture," Specification, 2018.

[4] F. Devic, L. Torres, and B. Badrignans, "Securing boot of an embedded linux on fpga," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*. IEEE, 2011, pp. 189–195.

[5] *IEEE Standard for Test Access Port and Boundary-Scan Architecture*, IEEE Std. IEEE Std 1149.1-2013, 2013.

[6] S. S. Ali, O. Sinanoglu, and R. Karri, "Test-mode-only scan attack using the boundary scan chain," in *2014 19th IEEE European Test Symposium (ETS)*. IEEE, 2014, pp. 1–6.

[7] S. Lapeyre, N. Valette, M. Merandat, M.-L. Flottes, B. Rouzeyre, and A. Virazel, "A lightweight, plug-and-play and autonomous jtag authentication ip for secure device testing," in *2022 IEEE European Test Symposium (ETS)*. IEEE, 2022.

[8] Arm Ltd., "Arm security technology—building a secure system using trustzone technology," White Paper, 2009.

[9] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*. ACM, 2020, pp. 1–16.

[10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 143–158.

[11] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys)*. ACM, 2008, pp. 315–328.

[12] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 379–394.

[13] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, 2014, pp. 1–14.

[14] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 207–220.

[15] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Transactions on Computer Systems*, pp. 1–70, 2014.

[16] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep. Technical Report 148, 1997.

[17] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang, "Llbt: An llvm-based static binary translator," in *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. ACM, 2012, pp. 51–60.

[18] A. Zakai, "Emscripten: An llvm-to-javascript compiler," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA Companion)*. ACM, 2011, pp. 301–312.

[19] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," Defense Technical Information Center (DTIC), Tech. Rep., 2014.