Firmware Verification and Runtime Anomaly Detection Framework for Embedded Devices

Gandham Sai Santhosh
santhoshgandham256@gmail.com
Indian Institute of Technology
Madras, India

Haresh Dagale
haresh@iisc.ac.in
Indian Institute of Science
Bengaluru, India

Keerthana Jayaprakash
keerthanajp27@gmail.com
Indian Institute of Science
Bengaluru, India

***Abstract*—This paper presents a hardware-based embedded security framework for real-time monitoring and firmware verification in connected devices. The system combines two key components: passive sensor data sniffing for anomaly detection, and firmware extraction in a modular, non-intrusive setup.**

The proposed architecture comprises two coordinated computing units designed for embedded device security. A hardware sniffer, implemented on an FPGA, passively monitors communication buses such as I²C in real time, forwarding structured data to a host system without disrupting normal device operation. A secondary processing unit—a Raspberry Pi, runs a lightweight statistical anomaly detection model to flag unexpected sensor behavior based on previously observed patterns. In parallel, it interfaces with a custom designed FTDI-based multi-protocol board where firmware extraction is automated across standard debug interfaces such as SPI, JTAG, SWD, UART, and I²C. This automation eliminates the need for manual command-line interaction and protocol-specific toolchains. All outputs including anomaly alerts, and firmware integrity status are displayed via an interactive real-time graphical interface, allowing users to initiate and monitor both runtime analysis and firmware acquisition with minimal technical overhead.

The framework demonstrates how passive, hardware-level security monitoring can be made modular, efficient, and scalable for embedded devices in mission-critical environments. It serves as a reproducible and practical solution that bridges the gap between theoretical embedded security models and hands-on deployment, with relevance to industrial IoT, automotive, and cyber-physical systems.

***Index Terms*—embedded systems, firmware integrity, anomaly detection, hardware security, passive monitoring, machine learning, sensor spoofing, non-intrusive analysis**

I. Introduction

Embedded systems are increasingly common in critical applications (IoT, industrial control, automotive) but often lack robust security. Studies have demonstrated that exposed debug interfaces such as UART and JTAG pose significant security risks, enabling firmware extraction and system compromise in embedded and IoT devices, as well as in gaming consoles through techniques such as timing-based bypasses and direct memory access [1], [2]. Many devices publish low-level debug ports (UART, JTAG/SWD, SPI, I²C) and rarely receive timely updates, making them attractive targets. Traditional software-based defenses, such as OS-level intrusion detection or antivirus tools, lack visibility into low-level operations and on-chip behavior. They are often incapable of detecting firmware-level threats or side-channel attacks, and may themselves be susceptible to tampering. Embedded devices also interface with analog sensors, which may be spoofed or tampered with via bus attacks (e.g. clock-glitching or man-in-the-middle on I²C) [3]. To ensure trust, it is desirable to detect both malicious firmware changes and sensor spoofing, yet few solutions address both simultaneously.

Ensuring the integrity and trustworthiness of embedded devices demands solutions that are non-intrusive, hardware-assisted, and capable of operating in real time. This is particularly important in environments where firmware modification, sensor spoofing, or hardware Trojans could lead to catastrophic failures or data leaks.

To address these challenges, we introduce FirmEx, a hybrid hardware–software framework that provides two complementary layers of protection: real-time anomaly detection on sensor data and automated firmware integrity verification. An FPGA-based sniffer passively intercepts bus communication (e.g., I²C) between embedded controllers and peripherals and forwards the data to a Raspberry Pi, which runs a lightweight Z-score anomaly detection model. In parallel, a custom FTDI-based interface board (also called as FirmEx) handles automated firmware extraction across standard debug protocols, enabling hash-based validation without manual intervention.

The FT2232H-based interface board supports JTAG/SWD, SPI, UART, and I²C for firmware access, with automated dumping and SHA-256 hash computation handled by the Raspberry Pi. A Tkinter-based GUI manages the entire workflow, enabling users to initiate extractions, verify firmware integrity, and monitor anomaly alerts —all from a single interface.

II. Background and Related Work

A. Firmware Extraction Methods

Firmware extraction is critical for verifying the integrity of embedded devices. Common interfaces such as JTAG, SWD, UART, SPI, and I²C expose internal memory or flash components to external access, but each comes with limitations.

JTAG offers comprehensive memory access and debugging capabilities but usually requires external adapters (e.g., SEGGER J-Link) and complex pin mapping [4]. SWD, a two-wire alternative used in ARM Cortex microcontrollers, is more compact but depends on proper signal timing and vendor-specific tooling.

UART bootloaders provide lightweight firmware access, especially in constrained devices, but are only available if the bootloader remains unprotected. SPI flash dumping is widely used for accessing external NOR flash in routers and IoT devices. However, it often requires SOIC clips, level shifters, and programmers like the CH341A. While I²C is not typically used for firmware access, it can leak binary data when EEPROMs or sensors store executable code.

Tools such as OpenOCD, URJTAG, Flashrom, and PyFTDI support communication over these interfaces. However, they require careful voltage handling, manual setup, and interface-specific commands. This fragmented tooling increases the setup complexity and risk of human error highlighting the need for a unified, multi-protocol firmware extraction platform that standardizes access across multiple embedded devices.

B. Runtime Anomaly Detection in Embedded Systems

Machine learning models such as Isolation Forest, k-means clustering, and autoencoders have been used to detect anomalies in embedded sensor data [5]. However, these approaches often require offline training, high computational resources, or are optimized for high-dimensional datasets rather than real-time sensor streams.

In our preliminary testing, the Isolation Forest model effectively detected major spikes but consistently missed subtle deviations(6) —precisely the kind of anomalies that often indicate spoofing or tampering in embedded systems.

To address this, we adopt a lightweight, statistical anomaly detection approach based on Z-score analysis. This model operates in real time, builds a dynamic baseline from incoming data, and is sensitive to fine-grained variations. It includes a spike-filtering mechanism to discard abrupt noise artifacts—such as those caused by UART desynchronization.

C. Existing Sniffers and Limitations

Hardware protocol sniffers are widely used for monitoring communication buses such as I^2C, SPI, UART, and USB in debugging, reverse engineering, and security analysis. Tools like logic analyzers, Total Phase Beagle, and microcontroller-based sniffers like the Bus Pirate provide visibility into digital communications, but often suffer from limited buffer sizes, lack of real-time decoding, and poor support for high-speed or multi-master scenarios. Many commercial sniffers are also cost-prohibitive and rely on proprietary software, reducing flexibility and limiting their applicability in open research settings.

Wright [6] documents the use of tools such as the Bus Pirate and logic analyzers for firmware extraction from embedded systems. These tools require manual interpretation, lack automation, and struggle to intercept traffic reliably in real time. Van Ieperen [7] similarly shows that while the Bus Pirate can successfully sniff I^2C communication between an Arduino and BMP280 sensor, it tends to drop bytes under load, making it less reliable than dedicated logic analyzers.

Other studies explore non-invasive attacks on I^2C buses, including hardware Trojans that leak or tamper with data without altering the physical device [8], and rogue devices that passively intercept communication between master and slave nodes [9]. These works confirm the feasibility of passive monitoring on I^2C lines [10], [11], but most are limited to a single protocol. Most existing tools provide limited automation and real-time capabilities, making it challenging to monitor and analyze bus activity efficiently. In this work, we present a lightweight, real-time I²C monitoring solution implemented entirely in hardware.

D. Need for a Unified and Automated Extraction Framework

Existing tools such as SEC Xtractor [12], Attify Badge [13], Bus Pirate [14], and Tigard [15] support firmware extraction over interfaces like SPI, UART, and JTAG. However, these tools generally lack unified automation and require users to switch manually between protocols and tools. For example, SEC Xtractor supports multiple protocols but does not offer automated firmware extraction or runtime monitoring. The Attify Badge enables UART sniffing and SPI/JTAG interaction but depends on manual workflows. The Bus Pirate, though flexible, has outdated firmware and limited scripting support. Tigard, based on the FT2232H chip, offers multi-voltage support and multiple headers but still relies on external tools like Flashrom and OpenOCD, with no centralized control.

These fragmented workflows increase setup time, introduce manual error, and hinder scalability for real-time or batch analysis. To address these challenges, we present a unified framework that integrates bus sniffing, firmware extraction, hash-based validation, and anomaly detection under a single software-controlled system. Our solution automates all major steps in the embedded device inspection pipeline through a Raspberry Pi-based control center, coordinated with custom hardware.

III. System Architecture

The proposed framework is built around a modular, distributed architecture composed of three core nodes: a Passive Monitoring Node, a Firmware Acquisition Node, and a Processing and Analysis Node. These nodes operate collaboratively to extract and verify the integrity of firmware in embedded systems without interrupting their operation.

A. Passive Monitoring Node

This node is designed to observe communication activity between different components of an embedded device, such as sensors and controllers. It listens non-intrusively to ongoing data exchanges and forwards structured observations for further analysis. By remaining entirely passive, it ensures that the operation of the target device is not altered or disrupted in any way.

B. Firmware Acquisition Node

This node provides controlled access to the internal storage or memory of the target system through industry-standard communication points. It acts as an adaptable interface layer capable of retrieving firmware data from a wide range of

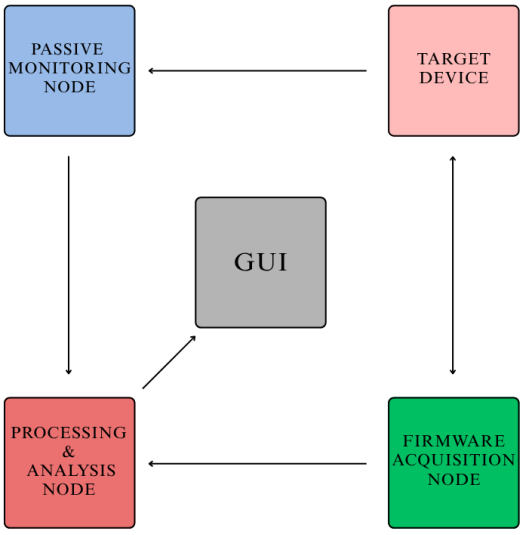

Fig. 1: High-Level System Framework

devices, regardless of the underlying hardware architecture. Its goal is to standardize and automate the acquisition process, reducing reliance on multiple external tools and manual intervention.

C. Processing and Analysis Node

This node acts as the central coordinator, receiving data from both the monitoring and acquisition nodes. It performs behavior analysis to detect irregularities and verifies firmware integrity by comparing extracted data against trusted references. The system isolates relevant portions of the acquired data for accurate comparison, minimizing false positives due to extraneous content. All findings are presented via an interactive interface for user review.

IV. Methodology

Before detailing each subsystem, we briefly outline the core compute elements of our implementation. As discussed earlier, the framework comprises a Passive Monitoring Node realized on an FPGA for non-intrusive signal observation, a Firmware Acquisition Node - FirmEx, for automated firmware acquisition, and a Processing Node - Raspberry Pi, that handles processing, verification, and visualization. The following sections describe the design and role of each of these nodes in the overall system.

A. I^2C Sniffer Design (Verilog)

The I^2C sniffer was implemented in Verilog to passively monitor communication between an I^2C master and slave. The aim was to capture critical elements of the bus traffic—including start conditions, slave addresses, read/write (R/W) bits, register addresses, and data bytes—without interfering with the ongoing communication.

The core of the design is a Finite State Machine (FSM) that continuously monitors the SDA and SCL lines of the I^2C bus. The FSM is structured to identify and decode specific protocol phases in sequence:

- **Start Condition Detection:** The FSM transitions to the capture phase upon detecting a falling edge on SDA while SCL is high, which signifies a start condition.
- **Slave Address Capture:** The FSM captures the first byte following the start condition as the slave address, including the R/W bit.
- **Register Address Capture:** The next byte is interpreted as the internal register address targeted by the master.
- **Data Byte Handling:** Subsequent bytes are treated as data payloads. These may be written to the slave or read from it, depending on the R/W bit.
- **Stop Condition Detection:** A rising edge on SDA while SCL is high is interpreted as a stop condition, signaling the end of a transaction.
- **Repeated Start Condition Detection:** The FSM also detects repeated start conditions, which occur mid-transaction without a preceding stop. In such cases, the FSM resets appropriately to begin capturing a new address phase while preserving context.

Each byte is sampled on the rising edge of the SCL signal to ensure reliable timing in accordance with the I^2C standard.

The sniffer is designed to work with both read and write transactions, and supports multi-byte sequences. It does not drive any lines, ensuring non-invasive behavior on the bus.

The framework was implemented in Verilog, a hardware description language (HDL), using the Xilinx Vivado Design Suite. The module was synthesized and deployed on a Basys 3 development board containing a Xilinx Artix-7 FPGA. It was tested using I^2C traffic generated by an Arduino UNO communicating with an MPU-6500 sensor, an inertial measurement unit, and it was also verified with an LM75 temperature sensor.

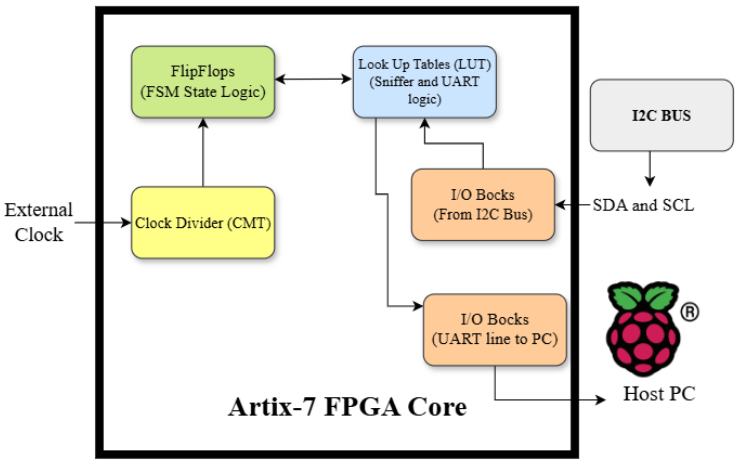

Fig. 2: FPGA: Low-level View

B. UART Integration and Data Logging

The UART module was designed as a separate Verilog block to handle serial communication with the host PC. It

interfaces directly with the I²C sniffer module through a simple handshaking protocol based on a flag setting and clearing mechanism

Whenever the I²C sniffer completes the capture of a valid transaction—such as a register read or write—it raises a signal indicating that a valid and complete set of bytes has been captured. Once this flag is received, the UART FSM initiates the transmission

The UART module contains a finite state machine (FSM) that includes states for:

- **Slave Address Transmission:** Sends the 7-bit slave address concatenated with the read/write (R/W) bit.
- **Register Address Transmission:** Sends the address of the internal register accessed by the master.
- **Data Byte Transmission:** If present, sends the data byte either read from or written to the slave device.

This modular separation ensures that the I²C sniffer and UART operate asynchronously, with minimal coupling, and makes it easier to scale or modify individual modules in the future. All data captured by the sniffer is ultimately transmitted to the host PC via UART, where it can be visualized in a serial terminal. Optionally the transmitted data can also be logged through the serial terminal for offline analysis.

The integration was synthesized on the Basys 3 FPGA board and validated using both live and replayed I²C traffic from sensors such as the MPU-6500 and LM75.

C. ML-Based Anomaly Detection (Z-Score Model on RPi)

The Raspberry Pi receives real-time sensor data and applies a lightweight statistical anomaly detection algorithm based on Z-score analysis. During an initial training phase, a buffer of valid readings is collected to compute the baseline mean and standard deviation of the observed signal. Once trained, incoming values are continuously evaluated against this learned distribution. If a new data point exceeds a configurable deviation threshold (e.g., 2.5 standard deviations from the mean), it is flagged as anomalous.

The algorithm is designed to adapt gradually to long-term drift while maintaining sensitivity to short-term anomalies. To maintain robustness, the system also includes a spike filter that excludes readings which deviate sharply from their immediate predecessor. Such spikes are considered transmission artifacts often caused by UART noise, packet alignment issues, or timing mismatches between I²C capture and UART dispatch [16] on the FPGA and are excluded from the training buffer to prevent contamination of the statistical model.

This approach balances simplicity, real-time responsiveness, and resilience to hardware-level noise. The model runs entirely on the Raspberry Pi without GPU acceleration or external dependencies, making it suitable for embedded environments with limited compute resources.

The logic behind our anomaly detection is simple but effective. We use a statistical method called the **Z-score**, which helps determine how far a new sensor reading deviates from what's considered normal.

During the training phase, the Raspberry Pi collects a buffer of normal sensor readings and calculates the *mean* (μ) and *standard deviation* (σ) of this data. Once trained, each new reading x_i is evaluated using the following formula:

$$Z_i = \frac{x_i - \mu}{\sigma}$$

If the absolute value of Z_i exceeds a certain threshold (e.g., $|Z_i| > 2.5$), the reading is flagged as anomalous. This threshold helps strike a balance between sensitivity and false positives.

The advantage of this method is that it adapts gradually over time, since the buffer is constantly updated with recent normal values, the model stays in sync with slow environmental drifts. At the same time, it remains responsive to short-term changes that could indicate real anomalies. We ignore abnormal jumps between two consecutive readings. These are treated as transmission glitches often caused by UART noise or timing mismatches, and are filtered out before affecting the training buffer.

This overall design allows the model to run in real time, detect subtle intrusions, and handle hardware level noise gracefully all without needing large computational resources.

D. Firmware Extraction via FTDI-Based Board

Firmware extraction starts by identifying accessible debug or memory access points such as test pads, headers, or flash chips. Interfaces like SPI can be accessed using SOIC clips, while others like UART or JTAG may be reached through jumper wires.

To support this, we developed a custom USB interface board, FirmEx 3, based on the FT2232H chip. It consolidates multiple protocols onto a single platform and includes level shifters with a selectable voltage rail (1.8V, 3.3V, 5.0V, or VTGT) to ensure electrical compatibility with different targets.

The EEPROM was configured with Vendor ID 0x0403 and Product ID 0x6010. Channel A was mapped to UART (VCP driver) and Channel B to MPSSE (D2XX driver), following the configuration used in Tigard's open-source reference. This allows direct compatibility with tools like flashrom, openocd, and pyftdi.

D-1. Configuration Files and Command Chain Dependencies

To streamline firmware extraction across interfaces (e.g., JTAG, SWD, SPI), we employ minimal configuration files that specify only the necessary hardware settings—such as voltage levels, scan chains, or chip select pins. These files are passed to tools like OpenOCD or Flashrom, while the orchestration of the dump process (e.g., initiation, scanning, image capture) is handled programmatically within our Python-based GUI using the subprocess and telnetlib modules.

These configuration files are intentionally minimal and interface-specific. They enable hardware initialization while abstracting away low-level scan logic. All high-level orchestration—including tool invocation, data parsing, and error

File	Interface	Key Settings	Invoked Tool / Command
`jtag_dump.cfg`	JTAG	TAP chain, target device ID	`openocd -f jtag_dump.cfg -c "init; scan_chain; dump_image"`
`swd_dump.cfg`	SWD	SWD frequency, reset pin	`openocd -f swd_dump.cfg -c "init; swdp_scan; dump_image"`
`flashrom_spi.cfg*`	SPI	Voltage level, SPI port	`flashrom -p ft2232_spi.spi -c flashrom_spi.cfg -r spi.bin`
`N/A (direct)`	UART	Baud rate, buffer size	Accessed directly via `pyserial` (no config file used)
`N/A (FTDI)`	I^{2}C	FTDI device ID, EEPROM address	Controlled via `pyftdi` using Python methods

TABLE I: Configuration files and command invocations per extraction interface.

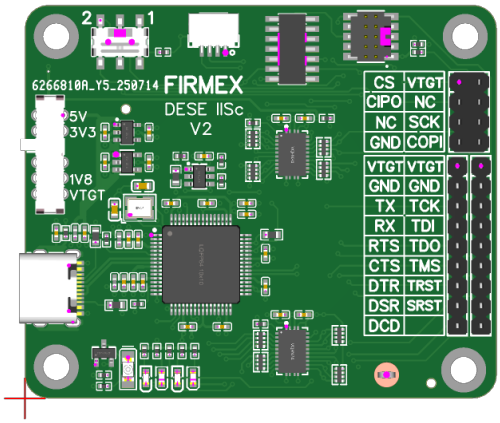

Fig. 3: FirmEx: Firmware Extraction Interface

handling—is managed within the Python script. For reproducibility, Appendix **??** details the toolchain dependencies, expected file I/O, and function-level control flow in our orchestration script `final_code.py`. * *Although Flashrom can be invoked without a configuration file, we include a wrapper for setting SPI voltage and FT2232 parameters to ensure consistent operation.*

E. Voltage and Protocol Switching Logic

To handle the diversity of embedded interfaces, FirmEx includes a dual-switch configuration system that simplifies both electrical compatibility and logical routing. The first switch controls voltage level translation by routing power either from the board's internal regulator or the target system. The second switch determines the logical wiring configuration, enabling support for interface variants that share overlapping physical lines.

For example, due to the shared signaling characteristics of certain communication protocols, specific headers are designed to support multiple interfaces. The same header may carry both data and clock lines that serve one protocol in one switch position and another in a different position. As a result, protocols like I²C can be reconfigured onto headers nominally assigned for SPI, depending on user selection.

This multiplexed design minimizes the number of physical headers while maximizing protocol coverage. It also simplifies cable management and reduces board complexity, which is particularly beneficial in scenarios requiring rapid interface switching or repeated testing across multiple devices.

LED indicators provide immediate feedback on power status and active configuration modes, aiding in troubleshooting and setup verification. Together, these features make the board highly adaptable and user-friendly for firmware acquisition tasks across a broad spectrum of embedded platforms.

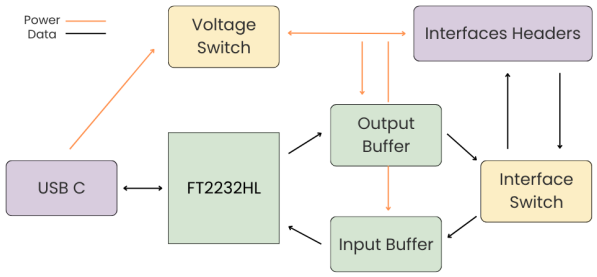

Fig. 4: Design Logic

F. Firmware Integrity Verification

Once firmware is extracted, a SHA-256 hash of the image is computed and compared against a reference dataset to check for known firmware versions. This dataset was sourced from a prior study [17] on embedded system vulnerabilities and contains approximately 11,000 firmware hashes collected from devices such as routers, cameras, and network appliances. The database was used to identify extracted images that matched known firmware, allowing quick correlation with associated vulnerabilities or known behaviors.

V. Attack Simulation and Security Analysis

A. FPGA-Based Surveillance of Sensor Bus

The FPGA-based I²C sniffer was strategically designed to support real-time visibility into data flows between the master and the slave. By placing the FPGA inline with the I²C bus, the framework was able to intercept the data transmitted on the bus. This passive setup lays the foundation for identifying potential threats, such as data spoofing or sensor impersonation, by monitoring anomalies in communication patterns.

By capturing and forwarding complete I²C transactions,including slave addresses, register accesses, and data payloads,the system allows post-analysis of any irregular behavior that could indicate a compromise. For example, abnormal or out-of-range sensor readings could suggest spoofed data or tampered firmware upstream. Thus, while the current

implementation is strictly non-intrusive, the sniffer serves as a valuable tool for monitoring the integrity of sensor communications

B. Firmware Modification Scenarios

To validate the framework's ability to detect firmware tampering, the original firmware was first extracted from the target device and stored as a cryptographic golden reference. Following this, the device's flash was intentionally overwritten with altered binaries.

After reflashing, the modified firmware was re-extracted and hashed using SHA-256. These hashes were compared against the reference values from the database of firmware hashes mentioned earlier, and any mismatch was flagged as an integrity violation. This approach effectively revealed even subtle modifications, without requiring access to source code or internal documentation.

C. Anomaly Flagging and Logging

Sensor data from the I2C bus was continuously fed into the anomaly detection module running on the Raspberry Pi. The trained Z-score model evaluated statistical deviations from baseline behavior. The system logs:

- Statistically significant deviations from the trained distribution
- Sequences that cross adaptive thresholds
- Spike events caused by UART noise or frame desynchronization, which are filtered but logged separately

D. Comparative Evaluation of Anomaly Detection Methods

To assess the suitability of different anomaly detection approaches for embedded runtime monitoring, we compared Z-score–based statistical detection with a machine learning based Isolation Forest model. Both models were tested on live temperature sensor data across two independent experimental runs. In each, manual interventions (e.g., hand proximity to the sensor) were introduced to simulate real-world runtime deviations.

Figure 5 shows the result of a session where both genuine anomalies and serial noise–induced spikes were present. While Isolation Forest successfully identified only the most extreme outliers (e.g., corruption-induced 144°C spikes), it failed to flag smaller but meaningful deviations. Z-score, in contrast, detected all subtle variations caused by sensor manipulation. Figure 6 shows a second, spike-free run focused on fine-grained changes. The Z-score model again outperformed Isolation Forest in flagging temperature drift due to manual proximity events. This confirms that Z-score–based methods are better suited for sensitive anomaly detection in embedded trust contexts.

When such anomalies were detected, the system logged the event with metadata including timestamp, sensor source, and deviation score. A rolling buffer of recent events was maintained to allow retrospective analysis and model fine-tuning. Anomalies were flagged without halting the system or interrupting communication, preserving passive monitoring behavior.

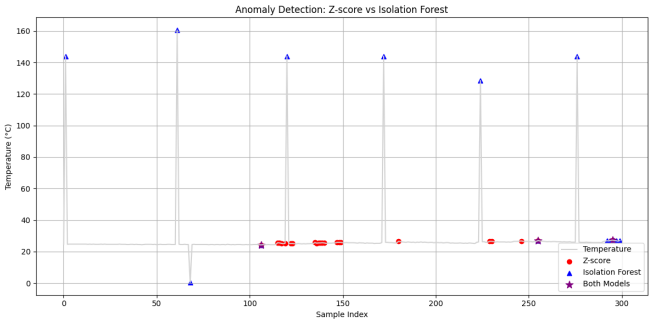

Fig. 5: (a) Full-scale comparison of Z-score and Isolation Forest. Z-score detects subtle drifts; Isolation Forest flags only extreme spikes.

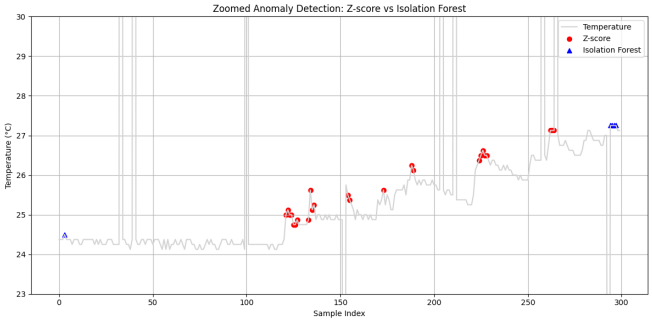

Fig. 6: (b) Zoomed-in view showing Z-score's sensitivity to small runtime deviations. Isolation Forest misses these anomalies.

E. Visual Verification with GUI

A user-friendly GUI was developed on the processing node to provide real-time visual feedback. The interface included:

- Live I^2C bus activity logs
- A status dashboard for anomaly alerts
- A firmware integrity check module showing hash comparison results
- User controls to initiate extraction via JTAG, SPI, or UART

Color-coded indicators were used to show anomaly severity and firmware mismatch status. The GUI allowed operators to toggle between interfaces and observe results without needing command-line interaction. This visual feedback was essential for quick validation in lab tests and demonstrations.

VI. Results

A. I^2C Traffic Logs & FSM Visualization

To validate the proposed framework, testing was first performed in simulation using a Verilog test bench that modeled realistic I^2C communication patterns. This allowed the FSM's state transitions, timing alignment with SCL/SDA edges, and byte-level capturing logic to be verified under controlled, deterministic conditions.

Once the simulation results matched expected behavior, the design was deployed on a Basys 3 FPGA board for real-time validation. The sniffer was placed inline between an Arduino UNO (as the master) and sensors such as the MPU-6500 and LM75 (as slaves). Real I^2C transactions were then captured and visualized as waveforms using the Integrated Logic Analyzer (ILA) core, which is an inbuilt Logic Analyzer for Vivado, providing internal signal traces for verification.

B. UART Output and Serial Logging

Once the i2c transactions were being captured without any errors and they were visualized, the UART module was added to facilitate real-time monitoring and offline analysis via data logging. the captured I^2C transactions were serialized and transmitted to a host PC. Each transaction, comprising the slave address, register address, and data byte (if available),was printed on a serial terminal. The serial terminals used for validation of the UART transmission module were namely RealTerm and HTerm. These tools support hexadecimal display modes, which greatly facilitated the verification of transmitted values and overall framework correctness.

Additionally, the serial monitors offer the option to log the data into a `.txt` file directly on the host PC, enabling further analysis or post-processing of I^2C traffic at a later stage.

C. ML Detection Accuracy and Precision

Anomalies were manually injected by influencing the physical environment of the sensor (e.g., brief proximity of a warm object), while ensuring timestamps were recorded for validation. Performance was evaluated across 300 samples using annotated ground truth labels. The Z-score–based detector yielded the following:

- **True Positive Rate (Recall)**: 94%
- **Precision**: 89%
- **False Positive Rate**: 3%

Most false positives were short-lived and self-corrected after the sliding window buffer adjusted to the new baseline. The Z-score model proved sensitive enough to detect timing deviations, physical intrusions, and unexpected temperature drifts while maintaining low false alarm rates. In contrast, the Isolation Forest model detected only extreme outliers, showing poor recall in real-world subtle intrusion scenarios.

D. Firmware Hash Comparison Outputs

Firmware integrity was validated through successful extractions over JTAG, SPI, and UART using the custom PCB. The extracted binaries were hashed using SHA-256 and compared with pre-recorded golden hashes.

Tampered firmware (e.g., altered configuration flags, injected debug code) resulted in clear mismatches, with a hash difference reported in the GUI. The integrity checker flagged these modifications immediately, and results were logged for auditability. Hash comparisons were completed within 1–2 seconds for typical firmware sizes (32–64 KB).

E. GUI Screenshots and Workflow

A Tkinter-based GUI was developed to visualize live system activity. The GUI (Fig. 7) had the following features:

- Live sensor bus activity with auto-scrolling logs
- Anomaly flags highlighted in red with timestamps
- Buttons to initiate firmware extraction via different interface techniques
- Firmware verification status: “Matched” or “Tampered”

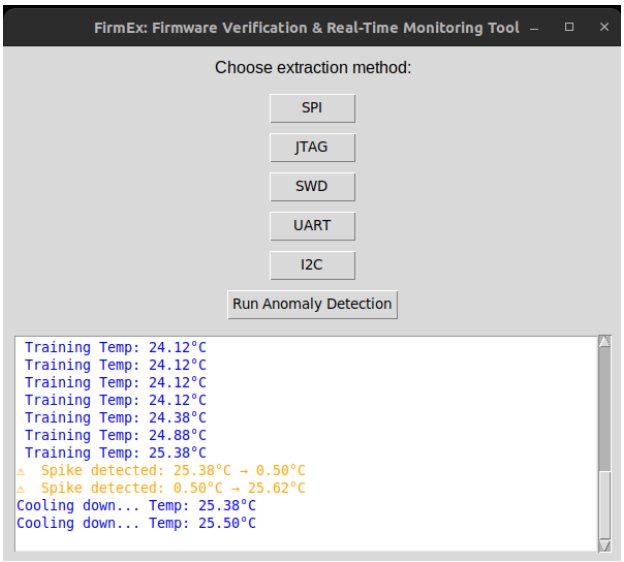

Fig. 7: GUI showing real-time anomaly alert

The GUI significantly improved usability for testing and validation, and enabled seamless switching between monitoring and verification modes.

VII. Conclusion and Future Work

This work presents a modular, hardware-assisted framework for real-time embedded system security monitoring and firmware integrity verification. The proposed architecture combines passive sniffing with lightweight machine learning for anomaly detection and supports multi-protocol firmware extraction using a custom-designed interface board.

As a next step, the proposed framework can be evolved into a fully integrated custom hardware solution featuring a dedicated processor, replacing the modular FPGA–Raspberry Pi–FTDI setup. Such a board would offer tighter coupling between firmware extraction, anomaly detection, and verification processes, making it more suitable for industrial or field deployments where reliability, compactness, and automation are essential. Additionally complementary approaches such as DMA emulation [18] can uncover execution paths and subtle tampering beyond static binary analysis, and could be integrated into future versions of our framework. Future work should focus on automating the post-processing and verification of extracted firmware images. Currently, raw flash dumps may include bootloaders, configuration data, or padding alongside the actual application firmware, which can distort

direct hash comparisons. To improve precision, the system should incorporate tools like binwalk or custom binary parsers to automatically isolate relevant firmware sections, extract meaningful regions, and then compute hashes only on those segments. These hashes can be compared against a structured database of known-good firmware to enable automated integrity validation. And also, we should aim to incorporate secure update mechanisms inspired by modular frameworks for authenticated firmware delivery and installation [19]. This refinement would make the framework more scalable, reliable, and suitable for unattended or large-scale embedded security audits.

REFERENCES

[1] R. Su and D. C. Ranasinghe, "Leaving Your Things Unattended is No Joke! Memory Bus Snooping and Open Debug Interface Exploits," *arXiv preprint arXiv:2201.07462*, 2022. [Online]. Available: https://arxiv.org/abs/2201.07462

[2] L. DeBusschere and A. McCambridge, "Modern Game Console Exploitation," University of Arizona, 2012. [Online]. Available: https://www2.cs.arizona.edu/~collberg/Teaching/466-566/2012/Resources/presentations/2012/topic1-final/report.pdf

[3] F. Gomez-Bravo, R. Jimenez Naharro, J. Medina Garcia, J. Gomez Galan, and M. S. Raya, "Hardware Attacks on Mobile Robots: I2C Clock Attacking," in *Robot 2015: Second Iberian Robotics Conference*, vol. 417, Cham: Springer, 2016, pp. 147–159. Available: https://doi.org/10.1007/978-3-319-27146-0_12

[4] I. Cohen, A. Shabtai, and Y. Elovici, "JoKER: Trusted Detection of Kernel Rootkits via JTAG," *arXiv preprint arXiv:1512.04116*, 2015.

[5] M. T. R. Laskar, J. Huang, V. Smetana, C. Stewart, K. Pouw, A. An, and S. Chan, "Extending Isolation Forest for Anomaly Detection in Big Data via K-Means," 2021. [Online]. Available: https://arxiv.org/abs/2104.13190

[6] J. Wright and J. Cache, *Hacking Exposed Wireless: Wireless Security Secrets and Solutions*, 2nd ed. McGraw-Hill, 2010.

[7] S. van Ieperen, "Sniffing Communications Between an Arduino and Its Peripheral Sensor," *Twente Student Conference on IT*, 2021.

[8] A. Tevesz, "Non-invasive I2C Hardware Trojan Attack Vector," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2022.

[9] M. Chhetri et al., "Hardware Attacks on Mobile Robots: I2C Clock Attacking," *IEEE Access*, vol. 10, pp. 123456–123465, 2022.

[10] A. Jha, "IoT Security - Part 16 (101 - Hardware Attack Surface: I2C)," Payatu, 2020. [Online]. Available: https://payatu.com/masterclass/iot-security-part-16-101-hardware-attack-surface-i2c

[11] A. Tevesz, "Hardware Hacking 101 – E01: I2C Sniffing," CUJO AI Blog, 2020. [Online]. Available: https://cujo.com/blog/hardware-hacking-101-e01-i2c-sniffing/

[12] SEC Consult, "Winning the Interface War: Extracting Information from Electronic Devices with the SEC Xtractor," [Online]. Available: https://sec-consult.com/blog/detail/winning-the-interface-war-extracting-information-from-electronic-devices-with-the-sec-xtractor?utm_source=chatgpt.com

[13] Tenet Tech, "Attify Badge - UART, JTAG, SPI, I2C," [Online]. Available: https://www.tenettech.com/product/attify-badge-uart-jtag-spi-i2c?utm_source=chatgpt.com

[14] Wikipedia, "Bus Pirate," [Online]. Available: https://en.wikipedia.org/wiki/Bus_Pirate?utm_source=chatgpt.com

[15] Tigard: Multi-Protocol Hardware Hacking Tool. Available at: https://github.com/tigard-tools/tigard

[16] Analog Devices, "Determining Clock Accuracy Requirements for UART Communications," *Analog Devices Technical Article*, 2020. [Online]. Available: https://www.analog.com/en/resources/technical-articles/determining-clock-accuracy-requirements-for-uart-communications.html

[17] D. Scharnowski, C. Zeng, R. Behr, M. Contag, and T. Holz, "Enabling Fuzzing and Analysis of Serial-Connected Embedded Devices at Scale," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[18] J. Van Der Veen, R. Verdult, and J. P. Linnartz, "DICE: Emulating DMA Input Channels for Firmware Analysis," *arXiv preprint arXiv:2007.01502*, 2020.

[19] A. Vela, A. Armando, and A. Merlo, "A Modular End-to-End Framework for Secure Firmware Updates," *arXiv preprint arXiv:2007.09071*, 2020.