Trust is Good, Monitoring is Better: FPGA- & TEE-Based Monitoring for Malware-Detection

Friederike Bruns*, Georg Gläser†, Florian Kögler†, Jonas Lienke†, Nithin R. Nanjundaswamy‡, Gregor Nitsche‡, Behnam R. Perjikolaei§, Jörg Walter§

*Carl von Ossietzky Universität Oldenburg, Germany, email: {first.last}@uol.de; †IMMS Institut für Mikroelektronik- und Mechatronik-Systeme gemeinnützige GmbH (IMMS GmbH), Ilmenau, Germany, email: {first.last}@imms.de; ‡DLR Institut für Systems Engineering, Oldenburg, Germany, email: {first.last}@dlr.de; §OFFIS e.V. Institut für Informatik, Oldenburg, Germany, email: {first.last}@offis.de

Abstract Ensuring trustworthiness in electronic systems is crucial to maintain safety and data integrity. Safety properties of robotic components are rigorously validated during development and, similarly, security requires ongoing monitoring during system operation as well. However, this monitoring must also safeguard its own components from tampering. We propose a novel runtime monitoring approach using application-specific monitors within an FPGA-based Trusted Execution Environment (TEE). To protect these monitors from supply chain attacks during design, fabrication, testing, or packaging, the TEE is programmed as the final step before deployment. The monitors are directly generated from formal constraint specifications established during the design and test phases. Our approach is demonstrated on a RISC-V-based System-on-Chip (SoC) for robotic applications, featuring a force sensor and a CAN-bus interface. We monitor the timing behaviour of hardware and software to detect malicious modifications affecting data transmission to a control unit. In an FPGA prototype, the monitors successfully identified hardware and software tampering. In real ASIC implementations, programming the TEE post-packaging ensures resilience against supply chain attacks.

This work was supported by the German Federal Ministry of Education and Research (BMBF) under funding codes 16ME0243K to 16ME0254.

1. Introduction

Ensuring the trustworthiness of electronic systems is increasingly critical due to risks from hardware faults, supply chain tampering, and environmental changes, which can compromise both functional and non-functional properties, such as timing or energy usage. Runtime verification can be implemented in software, hardware, or a combination of both. Software-based approaches, such as those by Do Tran et al. (2020); Havelund (2008); Tabakov & Vardi (2010), rely on finite-state machines or contract-based methods but introduce performance overhead by running alongside the system software. To address this, hardware-based solutions have emerged. For instance, Tracy et al. (2020) translate LTL formulas into FPGA automata, while Lu & Forin (2008) developed a compiler converting PSL specifications into Verilog. Decker et al. (2018) proposed real-time hardware-based monitoring on MPSoCs, and Nanjundaswamy et al. (2023) extended RISC-V to measure software execution time, though without considering execution time variability. Hybrid approaches, such as Solet et al. (2018), combine lightweight software instrumentation with hardware trace analysis but require variable mapping and incur significant overhead. Laurent et al. (2021) showed that hybrid techniques effectively detect hardware faults and software anomalies, especially in fault injection scenarios. However, most monitoring solutions neglect security threats like supply chain attacks or hardware Trojans, which Pan & Mishra (2022) highlight as critical concerns—especially AI-based Trojans that evade ML detection. Moreover, while runtime monitoring offers a way to detect deviations from nominal behaviour, it faces the same vulnerabilities as the systems it observes such as FPGA-based solutions like Rahman et al. (2024). Attacks may be introduced early in the design phase yet only manifest during operation, making detection challenging (Figure 1). This underscores the need for complementary, software-independent monitoring strategies. Compared to existing methods, our solution uniquely targets both functional faults and malicious modifications (e.g., HW-Trojans or firmware tampering), covers the full system lifecycle, and supports secure updates. The logical and physical isolation of monitors ensures their integrity, even in compromised environments.

We identify three core requirements: (i) monitoring both hardware and software behaviour, achievable only with hardware-based monitoring; (ii) updateability to adapt to changing system software; and (iii) logic isolation to ensure monitors cannot be detected or compromised by the system under observation. Potential threats considered include trojans that manipulate the software or hardware implementation and will lead to unexpected behaviour such as leaking internal data over communication links or executing additional functions, which will inevitably lead to different execution timing. While this work focuses on timing behaviour, approaches that observe the functional behaviour could be integrated in a similar way. We propose a dual-domain monitoring approach in Section 2. Lightweight hardware monitors are automatically generated from timing constraints, while firmware is observed using Statistical Timing Monitors (STMos), which compare execution time against learned timing profiles. To counter supply chain attacks, we embed an on-chip FPGA-based Trusted Execution Environment (TEE) for secure, out-of-band monitoring in Section 3. Our method is validated on a System-on-Chip with a RISC-V CPU and a manipulated CAN subsystem, where our monitors successfully detect malicious modifications and restore system integrity. Moreover, we compare our approach against existing techniques. Lastly, we discuss our results in Section 4.

2. Monitor Definition & Integration

We propose a runtime monitoring approach based on formally specified system behaviour, such as assume-guarantee contracts for timing properties using the MULTIC Timing Specification Language (MTSL) Böde et al. (2019). However, other specification languages or temporal logics could be used in a similar manner to express contracts, for example to adress functional behaviour. These constraints define the expected behaviour and are checked by monitors running inside a TEE, ensuring integrity even in compromised systems (Figure 1). One concern is the corruption of the monitors themselves, because a typical production flow consists of many supply chain participants, which can all be attack vectors. Figure 2 shows how our monitors are deployed post-production. Unlike conventional supply-chain-integrated monitors, ours are deployed post-production, enabling detection of tampered timing behaviour. This resides on two underlying assumptions: the trusted execution environment always executes monitors correctly, and there is a secure deployment/upgrade path for the monitors themselves. This is addressed via secure boot and cryptographic signatures.

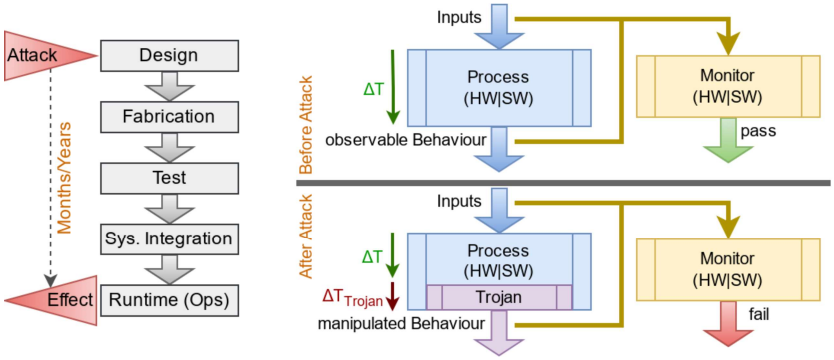

Figure 1. Monitoring (e.g. timing-fingerprints) to identify trojans that could be introduced at any stage of the supply chain.

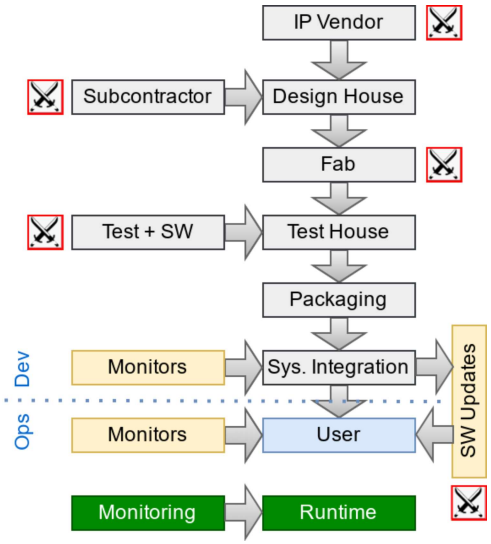

Figure 2. Integration of monitors at the end of development, bypassing multiple attack vectors (crossed swords).

Moreover, we distinguish between hardware and software monitoring, due to their differing timing characteristics:

For hardware, monitors operate at full clock speed to detect even single-cycle violations. Hardware IP-Monitoring (left Figure 3) includes event-specific observers that forward relevant signals to monitors, which verify compliance

with MTSL constraints. Depending on the hardware, observers are combinatorial or sequential (considering different signals) and, thus, it is impossible to give a generic definition of event observers and their logic. They are design specific. The MTSL verification algorithm within each hardware monitor includes a violation detection mechanism. Monitoring starts at event-triggered such as system startup or environmental stimuli at time $t = 0$ (representing physical time in clock cycles). Then, proceeds through Idle and Active states, where each subsequent request in the *Active* state resets the time to indicate the start of a new monitoring cycle. After evaluation of the timing condition and upon a detected violation, the monitor decides how to proceed. The monitors can handle recoverable (e.g., noisy input) and unrecoverable (e.g., critical violations) failures. This high-resolution, out-of-band monitoring enables low-overhead detection of rare timing anomalies, maintaining reliability. Consequently, Deviations from expected behaviour can be addressed promptly.

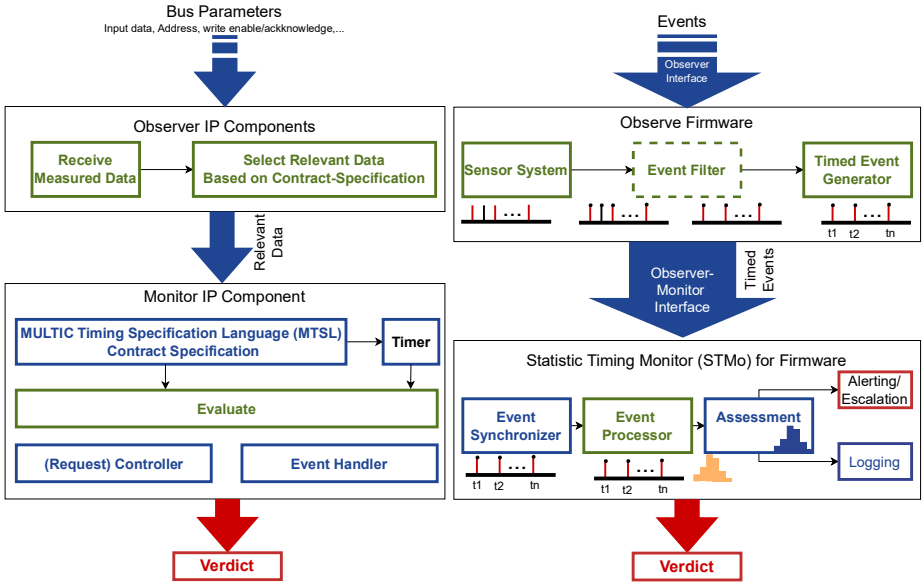

Figure 3. Monitoring hardware (left) and software (right).

On the other hand, timings for software on a modern CPU core are not necessarily fully deterministic, therefore we use a statistical approach. The developed Statistical Timing Monitors (STMo) system (Figure 3) effectively identifies modifications such as tampering that can affect functionality and security. Any modification (e.g., trojan insertion) will invariably change timing, which is captured as a shift Δt from the normal execution time t_{Norm}. Rather than relying solely on each single execution time, which can be influenced by factors like memory hierarchy, branch prediction, and I/O operations, STMo compares run-time timing statistics to a pre-defined histogram specification allowing for precise detection of software modifications. Here, an observer (integrated with a RISC-V core) timestamps software events using a synchronised clock. These timed events represent significant value- or state-changes of the software are filtered and passed to the STMo unit for synchronisation. The STMo builds a runtime histogram of execution timing and compares it to a reference histogram representing the expected behaviour within the assessment block. Deviations signal potential tampering and the assessment block triggers alerts or developer-defined recovery actions. All data is logged for traceability, enabling the back-tracing of any issues that may arise within the STMo system.

3. Case Study: Demonstrator ASIC

We present a RISC-V-based ASIC (Figure 4) designed for accurate force measurement in robotics, enabling collision detection via a Wheatstone bridge frontend and CAN interface. The chip includes a 32-bit RISC-V core, 16 kB SRAM, external QSPI NOR flash for program memory, and an eFPGA-based TEE attached via internal GPIOs for flexible runtime monitoring. The FPGA, built using the OpenFPGA toolchain Tang et al. (2019), features 200 LUTs to monitor processor activity. The FPGA bitstream is stored within the internal memory, allowing for rapid prototyping and testing.

For secure applications, access to the FPGA would need to be restricted —- either through JTAG or ROM-preloaded bitstreams. Fabricated in 180nm CMOS, the chip uses 155 886 standard cells (8.72 mm^2), with the FPGA accounting for 70 816 cells (1.5 mm^2) and 54% chip utilization. Estimated power consumption for the digital logic is 88 mW. A 'dummy Trojan' in the CAN-Controller demonstrates the monitor's capability to detect security threats effectively. We developed a custom firmware that retrieves and pre-processes data from a sensor before send to the robotic demonstrator's control unit via CAN, serving as a testbed for both hardware and software attack vectors.

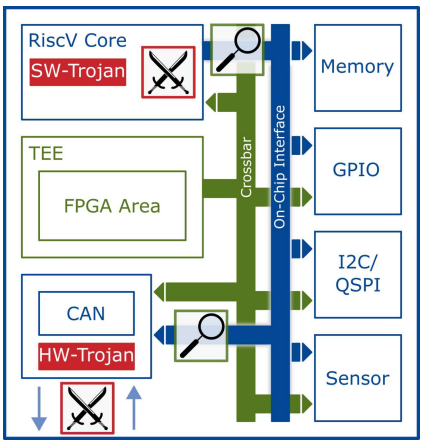

Figure 4. ASIC demonstrator with attack vectors (crossed swords) and monitors (lenses).

```
1  In each jumping window of length 100, with tolerance 0%:
2  reaction (RISC-V.GPIO.port[0]-port[7].0b00000001, RISC-V
       .GPIO.port[0]-port[7].0b00000010)
3  occurs within 39.08µs with symmetric jitter-distribution
       of {100% in [+0ns; -0ns]}
4   /* Example for jitter distribution */
5  symmetric jitter-distribution of {15% in [-1.5ns to -1.0
       ns];
6  20% in [-1.0ns to -0.5ns]; 30% in [-0.5ns to +0.5ns];
7  20% in [+0.5ns to +1.0ns];15% in [+1.0ns to +1.5ns];}
```

Listing 1: STMo specification for untampered firmware.

Figure 5. Untampered (top) vs. tampered (bottom) firmware.

Figure 6. Traces for a CAN IP transmission process where the data transmitted by cs_can2 trojanises cs_can in order to start an un-acknowledged sending process (tx_i).

To evaluate the STMo system, we tested firmware designed to send two-byte sensor data via CAN. A tampered version appended two extra bytes, simulating data exfiltration. The STMo system uses a baseline timing profile, derived from the untampered firmware, as a reference. Timing specifications are defined using Statistic Time Monitoring Specification Language (STMoSL), a domain-specific language for histogram-based constraints. While this paper does not provide a full overview of its syntax and semantics, yet, its features are exemplified by the specification of the untampered firmware. For instance (Listing 1), a jumping window of 100 executions with 0% tolerance captured consistent execution times of 39.08 µs (1954 clock cycles). This means, that after every 100 measurements the STMO system compares the measured run-time histogram with the STMoSL specification. STMoSL can also be used to specify timing distributions, e.g. due to jitter, using percentages of the number of measurements, and lower and upper interval-bounds relative to the previously specified mean. Characteristic events are specified by the port- & value-tuples in the reaction-phrase, referring to the RISC-V GPIO ports [0:7], and generated by the firmware using software annotation. Simulation with

both firmware versions showed STMo correctly validating the untampered case with verdict TRUE (top of Figure5) after 100 executions and flagging deviations in the tampered case with verdict FALSE (bottom of Figure5), proving its effectiveness in detecting timing discrepancies.

Hardware monitors and observers are implemented on the ASIC SoC FPGA Core (TEE), using GPIOs to capture CAN-Controller events. The objective is to avoid altering the original source code of the CAN-Controller in the ASIC design. We tested detection of a Hardware Trojan (HW-Trojan) integrated as a separate entity into the original CAN-Controller IP and controllable via a trojan_en input. Thus, Trojan logic can be bypassed when trojan_en is set to '0' without modifying the original CAN-Controller source code. The HW-Trojan listens on the CAN interface and, when triggered by a magic CAN packet, transmits previously recorded secret data (e.g. a cryptographic key) without authorisation or intervention by the software. *Aging* Monitors check timing and causality between request (event f) and send (event e). If a send occurs without a preceding request, the monitor flags a violation because of an MTSL aging pattern that specifies the backward delay: **Whenever** EventExpr **occurs then** EventExpr **has occurred within** Interval. Note that the interval detected by the pattern initiates upon the observation of the final element of an EventExpr. A waveform (Figure 6) shows two transmissions from cs_can2 to cs_can (tx2_i), where the second includes a Trojan activation code (8'h56) in register 8'd12. In the waveform, send_data_o and observer_go_tx are inputs to the cs_can monitor, while its outputs are o_pass and o_fail representing the monitor's verdict. To transmit the two sets of data the cs_can receives the send requests send_data_o2. With the trojanised data packet, cs_can transmits data (tx_i) without a proper send request. The monitor detects unauthorised transmission – send action (observer_go_tx = '1') occurs without a corresponding send request (send_data_o = '0') – and signals a violation (o_fail = '1'). This validates the monitor's ability to catch hardware-level threats.

Our runtime monitoring approach distinguishes itself from the related work by addressing multiple system vulnerabilities, including both hardware Trojans and malicious firmware, going beyond traditional software-focused solutions in terms of **target attacks** such as those by Tabakov & Vardi (2010); Havelund (2008). Unlike prior methods that primarily detect software faults, our solution provides comprehensive protection across hardware and software layers. Regarding the **time point of monitoring**, traditional approaches often focus solely on runtime monitoring, which can introduce latency and overlook hardware-level threats. In contrast, FPGA-based monitors allow us to use a uniform formal A/G contract specification across the entire life cycle. This ensures threats are detected early and reliably in both hardware and software **domains**, without incurring significant performance penalties. A major strength of our solution lies in the security and flexibility of the monitoring infrastructure: Isolation via TEE ensures the monitor's integrity, even if the main system is compromised. Dynamic **updatability** allows our monitors to evolve with emerging threats -— an advantage over fixed-function monitors such as those by Tracy et al. (2020). **Automated monitor generation** from constraint specifications accelerates deployment while maintaining correctness and security. Our monitors can observe both timing and data behaviour securely within the TEE (based on the constraint specification). Since individual participants of the supply chain are unaware of what's being observed, the risk of tampering is minimised enhancing **security**. This **logical separation** protects the monitor even in compromised environments. Our hybrid approach offers comprehensive runtime verification that ensures secure, flexible, and efficient monitoring, strengthening system resilience against both hardware and software vulnerabilities.

4. Conclusion

Unintended runtime behaviour in embedded systems, such as data leakage or malfunctions, can result from hardware Trojans or malicious firmware which can be introduced during design or manufacturing but which are often detected much later. We address this by a monitoring approach covering both hardware and software, and using TEEs for secure, out-of-band monitoring. The key features include comprehensive monitoring across hardware and software, updateability through automated constraint-based monitor generation, and logic isolation to ensure security even in compromised systems. Lightweight hardware monitors, isolated within FPGA-based TEEs, focus on timing behaviour, while STMoS handle software anomalies. The approach was successfully validated in an SoC with a RISC-V-based CAN-Controller,

detecting hardware and software tampering, including data exfiltration and altered data streams. This hybrid solution offers robust, real-time system integrity protection.

Future work could expand on this approach regarding the detection capabilities to cover more sophisticated threats, such as AI-based hardware Trojans or multi-vector attacks that combine both hardware and software vulnerabilities. Moreover, scalable deployment of these hybrid monitors could be addressed considering highly distributed systems, such as in automotive networks or industrial control systems, where centralised monitoring may not be feasible.

References

Böde, E., Damm, W., Ehmen, G., Fränzle, M., Grüttner, K., Ittershagen, P., Josko, B., Koopmann, B., Poppen, F., Siegel, M. & Stierand, I. (2019), MULTIC-Tooling, *in* 'FAT-Schriftenreihe 316', Forschungsvereinigung Automobiltechnik e.V. (FAT).

Decker, N., Dreyer, B., Gottschling, P., Hochberger, C., Lange, A., Leucker, M., Scheffel, T., Wegener, S. & Weiss, A. (2018), Online Analysis of Debug Trace Data for Embedded Systems, *in* 'Design, Automation and Test in Europe Conference and Exhibition (DATE)'.

Do Tran, D., Walter, J., Grüttner, K. & Oppenheimer, F. (2020), Towards Time-Sensitive Behavioral Contract Monitors for IEC 61499 Function Blocks, *in* 'IEEE Conference on Industrial Cyber Physical Systems (ICPS)'.

Havelund, K. (2008), Runtime Verification of C Programs, *in* '20th International Conference on Testing of Software and Communicating Systems', Springer-Verlag, Berlin, Heidelberg.

Laurent, J., Deleuze, C., Pebay-Peyroula, F. & Beroulle, V. (2021), 'Bridging the Gap between RTL and Software Fault Injection', *J. Emerg. Technol. Comput. Syst.* **17**(3).

Lu, H. & Forin, A. (2008), 'Automatic Processor Customization for Zero-Overhead Online Software Verification', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **16**.

Nanjundaswamy, N. R., Nitsche, G., Poppen, F. & Grüttner, K. (2023), RISC-V timing-instructions for open time-triggered architectures, *in* '53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)', IEEE, Porto, Portugal.

Pan, Z. & Mishra, P. (2022), Design of AI Trojans for Evading Machine Learning-based Detection of Hardware Trojans, *in* 'Design, Automation & Test in Europe Conference & Exhibition (DATE)'.

Rahman, M. M. M., Tarek, S., Azar, K. Z., Tehranipoor, M. & Farahmandi, F. (2024), 'The road not taken: efpga accelerators utilized for soc security auditing', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* .

Solet, D., Pillement, S., Béchennec, J.-L., Briday, M. & Faucou, S. (2018), HW-based Architecture for Runtime Verification of Embedded Software on SoPC systems, *in* 'NASA/ESA Conference on Adaptive Hardware and Systems (AHS)'.

Tabakov, D. & Vardi, M. Y. (2010), Optimized Temporal Monitors for SystemC, *in* H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky & N. Tillmann, eds, 'Runtime Verification', Springer-Verlag, Berlin, Heidelberg, pp. 436–451.

Tang, X., Giacomin, E., Alacchi, A., Chauviere, B. & Gaillardon, P.-E. (2019), OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs, *in* '29th International Conference on Field Programmable Logic and Applications (FPL)'.

Tracy, T., Tabajara, L. M., Vardi, M. & Skadron, K. (2020), Runtime Verification on FPGAs with LTLf Specifications, *in* 'Formal Methods in Computer Aided Design (FMCAD)'.