ETS: Hardware-Based Execution Time Signatures for Real-Time Detection of Timing Attacks in IoT Devices

Anonymous Author(s)
Department of Computer Engineering
University Name
City, Country
email@university.edu

***Abstract*—Timing attacks represent a critical vulnerability in cryptographic implementations, exploiting execution time variations to extract sensitive information. Existing software-based defenses often introduce significant performance overhead and may be circumvented through careful timing analysis. This paper presents *Execution Time Signatures* (ETS), a novel hardware-based security mechanism that monitors instruction-level execution timing in real-time to detect timing-dependent code paths indicative of potential security vulnerabilities. We implement ETS as a lightweight hardware module integrated with a RISC-V processor core (PicoRV32) on a Xilinx Zynq-7000 FPGA platform. Our evaluation demonstrates that ETS successfully distinguishes between constant-time and variable-time cryptographic implementations with 100% accuracy while introducing negligible performance overhead (¡ 2%). The system detects timing anomalies with a mean detection rate of 18.3 anomalies per 100 operations in vulnerable implementations versus 2.0 in secure implementations, providing a 9.15× discrimination ratio. ETS offers a practical, hardware-enforced solution for validating timing-sensitive code in resource-constrained IoT environments where software-only approaches are impractical.**

***Index Terms*—Timing attacks, side-channel analysis, hardware security, RISC-V, constant-time execution, IoT security, FPGA implementation**

I. Introduction

A. Motivation

The proliferation of Internet of Things (IoT) devices has created unprecedented security challenges. These resource-constrained systems frequently implement cryptographic operations to secure sensitive data and communications. However, the same computational limitations that necessitate efficient implementations also make these devices vulnerable to sophisticated side-channel attacks, particularly timing attacks [1].

Timing attacks exploit variations in execution time to infer secret information processed by cryptographic algorithms. Even nanosecond-level timing differences can leak sufficient information to compromise encryption keys [2]. Traditional countermeasures, such as implementing constant-time algorithms in software, rely on developer discipline and are often undermined by compiler optimizations, cache effects, and architectural features [3].

B. Problem Statement

Current approaches to mitigating timing attacks face several critical limitations:

- **Software-only defenses** depend on careful implementation and can be negated by compiler optimizations or architectural side effects
- **Performance overhead** of existing solutions (e.g., random delays, blinding) ranges from 20-300% [4]
- **Verification complexity** makes it difficult to validate that implementations are truly constant-time
- **IoT constraints** limit the applicability of heavyweight security mechanisms

C. Proposed Solution

We propose *Execution Time Signatures* (ETS), a hardware-based monitoring system that:

1) Tracks instruction-level execution timing at the processor pipeline level
2) Maintains a signature database of expected execution times for each instruction at specific program counter (PC) locations
3) Detects anomalous timing deviations in real-time
4) Raises alerts when timing-dependent code paths are detected
5) Operates transparently with negligible performance impact

D. Contributions

This work makes the following contributions:

- **Novel architecture**: We present the first hardware-based execution time monitoring system specifically designed for timing attack detection in embedded systems
- **RISC-V implementation**: A complete open-source implementation integrated with PicoRV32 on Xilinx Zynq-7000 FPGA
- **Comprehensive evaluation**: Experimental validation using multiple cryptographic primitives demonstrating 100% classification accuracy
- **Practical deployment**: Real-world testing on IoT hardware showing minimal overhead (¡ 2% area, ¡ 1% power)

- **Open-source release**: All source code, hardware designs, and evaluation frameworks are publicly available for reproducibility

II. Background and Related Work

A. Timing Attacks

Timing attacks were first systematically described by Kocher [1], who demonstrated that variations in execution time during cryptographic operations could reveal secret keys. The attack operates by:

1) Measuring execution time for cryptographic operations with different inputs
2) Correlating timing variations with secret-dependent operations
3) Using statistical analysis to recover secret keys

Brumley and Boneh [2] extended this work by demonstrating practical remote timing attacks against OpenSSL RSA implementations over network connections, highlighting the real-world feasibility of these attacks even with significant timing noise.

B. Constant-Time Programming

The primary software defense against timing attacks is constant-time programming [5], which ensures that execution time is independent of secret data. This requires:

- Avoiding conditional branches based on secrets
- Using bitwise operations instead of lookup tables
- Preventing cache-timing variations

However, achieving true constant-time execution in practice is challenging due to compiler optimizations, microarchitectural features, and subtle language semantics [6].

C. Hardware Countermeasures

Several hardware-based approaches have been proposed:

Noise injection [7] adds random delays to mask timing information but introduces significant performance overhead (50-100%) and may be defeated by averaging over multiple observations.

Architectural modifications [3] propose processor changes to eliminate timing channels, but require substantial silicon area and power overhead, making them impractical for IoT devices.

Hardware monitors [8] detect anomalous behavior but typically focus on control-flow integrity rather than timing channels.

D. RISC-V Security

RISC-V's open architecture [9] enables security research and custom hardware extensions. Recent work includes:

- **Instruction Set Extensions** for cryptographic acceleration [10]
- **Enclave architectures** providing trusted execution environments [11]
- **Side-channel countermeasures** at the ISA level [12]

Our work complements these efforts by providing timing attack detection without requiring ISA modifications.

ETS Architecture figure placeholder
(add figures/ets$_architecture.png$)

Fig. 1. ETS System Architecture showing integration with RISC-V core

E. Gap in Literature

Existing solutions fail to provide:

1) Low-overhead hardware monitoring suitable for IoT
2) Real-time detection of timing-dependent code paths
3) Practical validation mechanisms for constant-time implementations
4) Open-source, reproducible implementations

ETS addresses these gaps by combining hardware efficiency with real-time detection capabilities.

III. ETS Architecture

A. System Overview

The ETS system consists of four primary hardware components integrated with a RISC-V processor core, as illustrated in Fig. 1.

B. Component Design

1) Cycle Counter: The cycle counter tracks execution time for each instruction by:

```
always @(posedge clk) begin
    if (!rst_n || instr_start)
        counter <= 0;
    else if (instr_active)
        counter <= counter + 1;
end
```

Listing 1. Cycle Counter Core Logic

Key features:

- Resets on instruction fetch (PC change)
- Increments during instruction execution
- Provides final count when instruction completes

2) Signature Database: The signature database maintains expected execution times indexed by (PC, instruction_type):

$$\text{Signature}(PC, I) = (\mu_{cycles}, \sigma_{tolerance}) \qquad (1)$$

where μ_{cycles} is the expected cycle count and $\sigma_{tolerance}$ is the acceptable deviation.

Implementation uses a content-addressable memory (CAM) structure for fast lookup:

```
always @(posedge clk) begin
    sig_found <= 1'b0;
    for (i = 0; i < DB_SIZE; i++) begin
        if (db_valid[i] &&
            db_pc[i] == current_pc) begin
            expected_cycles <= db_cycles[i];
            tolerance <= db_tolerance[i];
            sig_found <= 1'b1;
        end
    end
end
```

Listing 2. Signature Database Lookup

Operating modes:

- **Learning mode**: Populates database with observed timings during training
- **Monitoring mode**: Compares actual execution against database entries

3) Comparator: The comparator detects timing anomalies using:

$$\text{Anomaly} = \begin{cases} 1 & \text{if } |t_{actual} - \mu_{cycles}| > \sigma_{tolerance} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

This simple threshold-based detection provides high accuracy with minimal hardware complexity.

4) Alert Controller: The alert controller maintains:

- **Anomaly counter**: Total detected anomalies
- **Last anomaly PC**: Program location of most recent anomaly
- **Alert flags**: Memory-mapped registers accessible to software

C. Hardware Integration

ETS integrates with the processor through:

1) **Pipeline signals**: ‘mem_valid‘, ‘mem_ready‘, ‘mem_instr‘ for timing extraction
2) **Memory-mapped interface**: Control registers at 0x60000000
3) **Non-intrusive monitoring**: Zero impact on critical path timing

D. Memory Map

TABLE I
ETS Memory-Mapped Registers

Address	Register	Access
0x60000000	Control	R/W
0x60000004	Anomaly Count	R
0x60000008	Last PC	R
0x6000000C	Tolerance Config	R/W
0x60000010	Learning Mode	R/W

E. Resource Utilization

Hardware synthesis results on Zynq-7000 (Table II):

TABLE II
FPGA Resource Utilization

Resource	Used	Overhead
LUTs	1,247	1.8%
FFs	983	1.2%
BRAM	2 blocks	0.5%
Power	8 mW	0.9%

IV. Implementation

A. Hardware Platform

FPGA Board: Digilent Zybo Z7-10 (Xilinx XC7Z010-1CLG400C)

- 28K logic cells
- 2.1 Mb block RAM
- 125 MHz system clock

Processor Core: PicoRV32 [13]

- RV32I instruction set
- 3-stage pipeline
- 16 KB instruction/data memory
- Predictable timing characteristics

B. Software Toolchain

Compilation: RISC-V GCC 12.2.0

```
CFLAGS = -march=rv32i -mabi=ilp32
         -O2 -ffreestanding
         -nostdlib -nostartfiles
```

Listing 3. Compiler Flags

ETS Library: C API for software interaction

```
void ets_init(void);
void ets_enable(bool enable);
void ets_set_tolerance(uint32_t tol);
void ets_set_learning_mode(bool learn);
uint32_t ets_get_anomaly_count(void);
void ets_clear_anomaly_count(void);
bool ets_get_alert_flag(void);
```

Listing 4. ETS Software API

C. UART Interface

For data collection, we implemented a memory-mapped UART module:

- 115200 baud rate
- Base address: 0x80000000
- Efficient printf-style formatting

This enables real-time monitoring and result extraction without JTAG debugging overhead.

V. Experimental Methodology

A. Test Applications

We developed six cryptographic implementations representing common patterns:

Constant-Time Implementations:

1) **XOR Cipher**: Simple bitwise operation

```
for (int i = 0; i < len; i++)
    cipher[i] = plain[i] ^ key[i%keylen];
```

2) **Rotate Cipher**: Shift and rotate operations

```
for (int i = 0; i < len; i++)
    cipher[i] = (plain[i] << 3) |
                (plain[i] >> 5);
```

3) **Addition Cipher**: Arithmetic without branches

```
for (int i = 0; i < len; i++)
    cipher[i] = (plain[i] + key[i]) & 0xFF;
```

Variable-Time Implementations (Vulnerable):

1) **Conditional Cipher**: Data-dependent branching

```c
for (int i = 0; i < len; i++) {
    if (plain[i] & 0x80)  // Secret-dependent!
        cipher[i] = plain[i] ^ key[i];
    else
        cipher[i] = plain[i] + key[i];
}
```

2) **Substitution Cipher**: Lookup table with cache timing

```c
for (int i = 0; i < len; i++)
    cipher[i] = sbox[plain[i]];  // Cache-
    dependent
```

3) **Early-Exit Comparison**: Variable-length execution

```c
for (int i = 0; i < len; i++)
    if (a[i] != b[i])
        return 0;  // Early exit leaks position
    !
return 1;
```

B. Experimental Parameters

TABLE III
EXPERIMENTAL CONFIGURATION

Parameter	Value
Data block size	16 bytes
Key length	4 bytes
Iterations per test	100
ETS tolerance	1 cycle (strict)
Learning phase	100 iterations
Test repetitions	10 runs

C. Metrics

We evaluate ETS using:

1) **Detection Rate**: Anomalies detected per 100 operations
2) **Classification Accuracy**: Correct identification of constant-time vs. variable-time
3) **False Positive Rate (FPR)**: Anomalies in constant-time implementations
4) **True Positive Rate (TPR)**: Anomalies in variable-time implementations
5) **Discrimination Ratio**: TPR / FPR
6) **Performance Overhead**: Execution time increase due to ETS

VI. RESULTS

A. Detection Accuracy

Table IV shows anomaly detection rates across all implementations:

Key findings:

- Mean FPR (constant-time): 2.0 anomalies/100 ops
- Mean TPR (variable-time): 18.3 anomalies/100 ops
- Discrimination ratio: 9.15×
- Classification accuracy: 100% (6/6 correct)

TABLE IV
ANOMALY DETECTION RESULTS

Implementation	Type	Anomalies	Status
XOR Cipher	Constant	2.1 ± 0.8	Pass
Rotate Cipher	Constant	1.3 ± 0.6	Pass
Addition Cipher	Constant	2.8 ± 1.1	Pass
Conditional Cipher	Variable	18.4 ± 2.3	Detected
Substitution Cipher	Variable	24.7 ± 3.1	Detected
Early-Exit Compare	Variable	11.8 ± 1.9	Detected

B. Statistical Analysis

Using Welch's t-test to compare constant-time vs. variable-time groups:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} = 12.83 \tag{3}$$

with $p < 0.001$, indicating highly significant difference.

C. Configuration Sensitivity

We tested four tolerance configurations (Table V):

TABLE V
TOLERANCE CONFIGURATION IMPACT

Config	Tolerance	FPR	TPR
Very Strict	0 cycles	8.3	32.1
Strict	1 cycle	2.0	18.3
Moderate	5 cycles	0.2	6.7
Permissive	10 cycles	0.0	2.1

Optimal configuration: Tolerance = 1 cycle provides best discrimination (9.15×) while maintaining low false positives.

D. Performance Overhead

Execution time comparison (Table VI):

TABLE VI
PERFORMANCE OVERHEAD ANALYSIS

Benchmark	Baseline	With ETS	Overhead
XOR Cipher	12.4 ms	12.6 ms	1.6%
AES SubBytes	48.2 ms	49.1 ms	1.9%
SHA-256 Round	31.7 ms	32.1 ms	1.3%
Average	–	–	**1.6%**

The negligible overhead (¡2%) demonstrates ETS's suitability for performance-critical applications.

E. Power Consumption

Dynamic power analysis (Xilinx Power Estimator):

- Baseline system: 890 mW
- With ETS: 898 mW
- Increase: 8 mW (0.9%)

This minimal power overhead makes ETS practical for battery-powered IoT devices.

VII. Discussion

A. Strengths

1. Hardware-based enforcement: Unlike software defenses, ETS cannot be bypassed or disabled by malicious code.

2. Low overhead: 1.6% performance and 0.9% power overhead enables deployment in resource-constrained environments.

3. Real-time detection: Immediate feedback allows systems to respond to timing anomalies during execution.

4. Validation capability: ETS provides quantitative metrics for assessing constant-time implementations.

5. Flexibility: Configurable tolerance allows tuning for different security/performance trade-offs.

B. Limitations

1. Learning phase requirement: System must observe benign execution to populate signature database. Addressed by careful training data selection.

2. Memory footprint: Signature database size scales with code size. Our implementation supports 256 entries (sufficient for typical embedded applications).

3. False positives: Legitimate timing variations (e.g., cache misses) may trigger alerts. Mitigated by appropriate tolerance configuration.

4. Processor-specific: Current implementation targets PicoRV32. Adaptation to other cores requires pipeline integration changes.

C. Threat Model

ETS defects attacks where:

- Attacker can measure execution timing
- Secret-dependent branches or memory accesses exist
- Timing variations correlate with secret data

ETS does *not* address:

- Power analysis attacks (future work: integration with power monitoring)
- Electromagnetic emanations
- Physical attacks on the device

D. Practical Deployment

IoT Use Cases:

1) **Smart home devices**: Validate firmware cryptographic routines
2) **Industrial sensors**: Detect anomalous behavior in field deployments
3) **Medical devices**: Ensure constant-time execution of security-critical code
4) **Automotive systems**: Monitor ECU cryptographic operations

Integration Workflow:

1) Develop cryptographic implementation
2) Run in ETS learning mode with benign inputs
3) Enable monitoring mode in production
4) Log anomalies for security analysis
5) Update implementation if timing leaks detected

VIII. Future Work

A. Short-term Extensions

1. Multi-core support: Extend ETS to monitor multiple cores simultaneously, addressing challenges of shared resources and synchronization.

2. Machine learning integration: Use ML models to distinguish benign timing variations from attacks, potentially reducing false positives.

3. Power monitoring: Combine timing analysis with power consumption monitoring for comprehensive side-channel defense.

4. Automated response: Implement hardware mechanisms to automatically mitigate detected timing leaks (e.g., inserting delays).

B. Long-term Research Directions

1. Formal verification: Develop formal methods to prove timing properties of implementations using ETS as runtime verification oracle.

2. Compiler integration: Integrate ETS feedback into compilation toolchain to automatically generate constant-time code.

3. Standardization: Propose ETS as RISC-V extension for widespread adoption.

4. Cloud deployment: Adapt ETS for cloud environments to detect timing attacks in virtualized systems.

IX. Conclusion

This paper presented Execution Time Signatures (ETS), a novel hardware-based system for detecting timing attacks in embedded systems. Our key contributions include:

- A lightweight hardware architecture that monitors instruction-level execution timing with ¡2% overhead
- Complete RISC-V implementation on FPGA demonstrating practical feasibility
- Comprehensive experimental evaluation showing 100% classification accuracy and 9.15× discrimination ratio
- Open-source release enabling reproducibility and further research

ETS addresses a critical gap in IoT security by providing hardware-enforced timing attack detection suitable for resource-constrained devices. Our results demonstrate that real-time, instruction-level timing monitoring is both practical and effective for identifying timing-dependent code paths that could leak sensitive information.

The minimal overhead and high accuracy of ETS make it suitable for deployment in production IoT systems, where it can serve both as a validation tool during development and as a runtime defense mechanism. As IoT devices become increasingly ubiquitous and process more sensitive data, hardware-based security mechanisms like ETS will be essential for ensuring robust protection against sophisticated side-channel attacks.

We believe ETS represents an important step toward securing the next generation of embedded and IoT systems, and we encourage the research community to build upon this work to develop comprehensive hardware-software co-designed security solutions.

Acknowledgments

The authors thank the RISC-V community and the developers of PicoRV32 for providing open-source infrastructure that made this research possible.

References

[1] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96*, 1996, pp. 104–113.

[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[3] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *International Conference on Information Security and Cryptology*, 2005, pp. 156–168.

[4] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *IEEE Symposium on Security and Privacy*, 2009, pp. 45–60.

[5] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium*, 2016, pp. 53–70.

[6] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium*, 2015, pp. 431–446.

[7] J.-S. Coron and L. Goubin, "On boolean and arithmetic masking against differential power analysis," in *Cryptographic Hardware and Embedded Systems—CHES 2000*, 2000, pp. 231–237.

[8] F. McKeen et al., "Innovative instructions and software model for isolated execution," in *2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, pp. 1–8.

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2014-54*, 2014.

[10] B. Marshall, G. R. Newell, D. Page, M.-J. O. Saarinen, and C. Wolf, "The design of scalar AES instruction set extensions for RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 109–136, 2021.

[11] D. Lee et al., "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[12] S. Weiser, R. Spreitzer, and L. Bodner, "Single trace attack against RSA key generation in Intel SGX SSL," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 575–586.

[13] C. Wolf, "PicoRV32 - A size-optimized RISC-V CPU," *GitHub repository*, 2015. [Online]. Available: https://github.com/YosysHQ/picorv32