Preprints.org

Article

Not peer-reviewed version

Timing Side-Channel Leakage and Fault Resilience Evaluation of Lightweight Authentication on Arduino-Class MCUs

Ajay A. Waghmare * and Subramaniam Ganesan

Posted Date: 22 April 2026

doi: 10.20944/preprints202604.1551.v1

Keywords: embedded systems security; Automotive ECU Security; timing side-channel analysis; CAN network security; lightweight authentication; microcontroller security; arduino-based evaluation; implementation-level security; resource-constrained devices

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Timing Side-Channel Leakage and Fault Resilience Evaluation of Lightweight Authentication on Arduino-Class MCUs

Ajay A. Waghmare * and Subramaniam Ganesan

Department of Electrical and Computer Engineering, Oakland University, Rochester, Michigan, USA
* Correspondence: ajaywaghmare@oakland.edu

Abstract

Automotive electronic control units increasingly rely on resource-constrained microcontrollers to implement authentication and message validation for in-vehicle networks such as CAN. Although cryptographic primitives may be mathematically secure, their software implementations on low-cost ECUs can leak sensitive information through timing side-channels. This paper presents a practical timing side-channel evaluation of an embedded authentication routine implemented on Arduino-class microcontrollers representative of entry-level automotive ECUs. A Python-based measurement framework interacts with the target device over a serial interface, repeatedly triggers authentication operations, and records execution time under controlled conditions. Experimental results show statistically distinguishable timing distributions correlated with secret-dependent execution paths, enabling an attacker with CAN-adjacent access to infer partial information about authentication checks. Lightweight countermeasures, including constant-time comparisons and control-flow normalization, are implemented and evaluated. The mitigated design reduces observable timing leakage with minimal execution-time and memory overhead, highlighting the need for implementation-level timing analysis in automotive embedded systems.

Keywords: embedded systems security; Automotive ECU Security; timing side-channel analysis; CAN network security; lightweight authentication; microcontroller security; arduino-based evaluation; implementation-level security; resource-constrained devices

---

I. Introduction

Side-channel attacks constitute a well-established class of implementation-level attacks that exploit unintended information leakage arising from the physical or microarchitectural behavior of computing systems. Unlike classical cryptanalysis, which targets weaknesses in cryptographic algorithms, side-channel attacks leverage observable effects—such as execution-time variations, power consumption, or electromagnetic emanations—to infer sensitive information processed by a device. Among these, timing side-channel attacks exploit data-dependent execution-time variations and have been shown to enable practical recovery of cryptographic secrets from real implementations, even when mathematically secure algorithms are employed [1,2].

The majority of existing timing side-channel research has focused on high-performance processor architectures. Such systems expose complex microarchitectural features, including caches, speculative execution, branch prediction, and shared execution resources, which may retain persistent or transient footprints of victim execution. These features have been extensively studied as sources of timing leakage and have motivated a large body of work on cache-based and microarchitectural side-channels, as well as their corresponding defenses [3–6].

In contrast, microcontroller units (MCUs), which are widely deployed in embedded and automotive electronic control units (ECUs), are typically designed with simple, in-order processor cores and limited microarchitectural state. MCUs often lack caches, speculative execution, and

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

simultaneous multithreading, and they generally do not permit concurrent execution of attacker and victim tasks. Owing to these characteristics, MCU-based systems have historically been assumed to offer a higher degree of inherent resistance to timing side-channel attacks and, consequently, have received comparatively limited attention in the literature [7,8].

Recent studies have demonstrated that this assumption does not hold in general. Even in the absence of traditional microarchitectural buffers, MCUs may exhibit exploitable timing leakage originating from system-level interactions. In particular, shared on-chip resources such as buses, interconnects, memory arbitration logic, and peripheral intellectual-property blocks can introduce timing dependencies that persist across context switches or interrupt boundaries. These MCU-wide timing side-channels enable an attacker to influence or observe the execution behavior of a victim task despite the architectural simplicity of the processor core [9–13].

The implications of such vulnerabilities are especially significant in automotive and industrial embedded systems. Modern vehicles integrate numerous ECUs that execute security-critical functionality, including secure boot, diagnostic authentication, message authentication, and software-update verification. In these environments, an adversary may interact with an ECU through diagnostic interfaces, compromised gateway modules, or software vulnerabilities. Timing side-channel leakage at the MCU or system-on-chip (SoC) level therefore represents a realistic threat vector that does not require invasive physical access or specialized laboratory equipment [14,15].

Despite increasing recognition of timing side-channel risks in embedded systems, practical evaluation methodologies suitable for early-stage ECU software development remain limited. A substantial portion of existing work focuses on formal verification at the hardware-design level, assumes detailed knowledge of register-transfer-level implementations, or relies on specialized measurement infrastructure [16–19]. Recent survey studies further highlight the gap between theoretical side-channel models and practical evaluation on low-cost embedded platforms [20]. This paper addresses this gap by experimentally analyzing timing side-channel leakage on an automotive-representative microcontroller platform using low-cost and widely available tools. The contributions of this work are summarized as follows:

1. We demonstrate that measurable timing side-channel leakage can arise on low-complexity microcontrollers executing authentication routines representative of automotive ECU workloads.
2. We present a Python-based measurement framework that enables systematic timing analysis over standard communication interfaces, facilitating reproducible evaluation on resource-constrained platforms.
3. We evaluate lightweight software-level countermeasures and analyze their effectiveness and overhead in a real-time embedded context.

II. Timing Side-Channel Leakage and Fault Resilience

A. Threat Model

This work considers a software-based timing side-channel adversary targeting an embedded microcontroller unit (MCU) executing security-critical routines, such as authentication or key-verification functions, representative of automotive electronic control units (ECUs). The adversary is assumed to have the ability to repeatedly invoke the target function and to observe its execution time with sufficient resolution through software-accessible interfaces, such as serial communication, GPIO signaling, or network-mediated request–response timing. This attacker model is consistent with classical timing-attack formulations and their embedded extensions [1,2].

The adversary does not possess physical probing capabilities, invasive measurement equipment, or direct access to internal microarchitectural state. Instead, the attack relies solely on externally observable timing variations arising from data-dependent execution behavior. The attacker is assumed to have knowledge of the target algorithm and its implementation model, but not of secret values such as cryptographic keys or authentication tokens, in line with Kerckhoffs' principle [1].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

Unlike attacks on high-performance processors, the considered threat model does not assume concurrent execution of attacker and victim tasks or the presence of complex microarchitectural features such as caches, speculative execution, or simultaneous multithreading. Instead, timing leakage may arise from control-flow divergence, memory-access patterns, interrupt handling, or contention on shared system resources, even when tasks execute sequentially [3,4].

In the automotive context, this threat model reflects realistic attack surfaces in which an adversary interacts with an ECU through diagnostic services, compromised gateway modules, or externally accessible communication interfaces. Timing observations can be collected over multiple executions to reduce noise and extract statistically significant leakage, as demonstrated in both classical and MCU-specific timing-attack literature [1,5]. Denial-of-service attacks and permanent fault injection are outside the scope of this work; however, the interaction between timing leakage and software-level fault-resilience mechanisms is considered, as timing-dependent error handling may itself introduce additional leakage channels [6].

B. MCU Timing Side Channels in a Nutshell

Timing side channels arise when the execution time of a computation depends on secret-dependent data or control flow. Early work established that even small timing variations can be exploited to recover cryptographic secrets when measured across repeated executions [1]. While such attacks were initially demonstrated on cryptographic co-processors and general-purpose CPUs, similar principles apply to embedded MCUs despite their architectural simplicity [2].

Traditionally, MCUs were considered less susceptible to timing side-channel attacks due to their lack of caches, speculative execution, and parallel attacker–victim execution. However, recent work has shown that timing leakage can still emerge at the system-on-chip (SoC) level through shared resources such as buses, memory arbitration logic, interrupt controllers, and peripheral IP blocks [3,4]. These MCU-wide timing side channels persist across context switches and do not require concurrent execution, contradicting long-held assumptions about embedded-platform security.

From a software perspective, timing leakage on MCUs commonly originates from data-dependent branching, loop bounds, memory-access sequences, and error-handling paths. Authentication routines that perform byte-wise comparisons, conditional early exits, or variable-time retries are particularly susceptible [1,5]. Even when cryptographic primitives are constant-time in isolation, their integration into higher-level firmware logic may reintroduce timing variability.

Recent surveys and experimental studies emphasize that timing side channels in embedded systems often manifest as subtle but measurable variations that require careful statistical analysis rather than direct observation [2,7]. As a result, low-cost experimental setups using software-based timing measurements have proven sufficient to demonstrate practical leakage on MCU platforms [3,8]. These findings motivate systematic evaluation methodologies that can be applied during early-stage firmware development, prior to hardware deployment.

C. Experimental Timing Leakage Model

Figure 1 illustrates the experimental timing-leakage model considered in this work. The model captures a black-box interaction between an external requester and an MCU/ECU executing security-critical firmware, in which timing information is inferred solely from end-to-end request–response behavior, consistent with classical timing-attack threat models [1,2].

An external attacker, implemented as a Python-based timing analyzer, repeatedly issues requests to the target MCU through a standard communication interface. These requests invoke a security-relevant application-level function, such as authentication or key-verification logic, implemented in the ECU firmware. The attacker does not observe intermediate states or internal signals; instead, only the total response latency associated with each request is measured and recorded, reflecting a black-box observation model commonly assumed in timing side-channel analysis [7,11].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

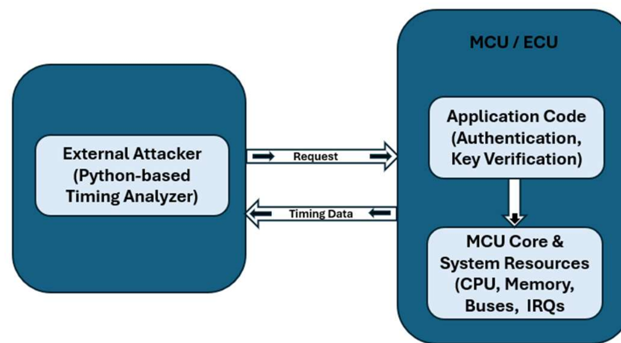

Figure 1. Timing Side-channel leakage model on an MCU.

Within the MCU/ECU boundary, the application code is executed by the MCU core and its associated system resources, including the processor pipeline, memory subsystem, on-chip buses, and interrupt-handling logic. The downward control flow depicted in Figure 1 reflects the fact that application-level instructions are fetched, decoded, and executed by the MCU core, with execution behavior directly influencing the utilization of shared resources. Any data-dependent control flow, memory-access pattern, or error-handling path within the application code may therefore introduce variability in execution time [3,9].

Timing leakage arises when secret-dependent behavior in the application code propagates to observable execution-time variations at the MCU core or system-resource level. Even in the absence of caches or speculative execution, such variability may result from conditional branches, loop bounds, memory-arbitration delays, interrupt servicing, or peripheral interactions. Recent work has shown that these MCU-wide timing side channels can persist despite architectural simplicity [9,12].

The timing-data arrow in Figure 1 represents the aggregation of these internal effects into a single end-to-end latency measurement. From the attacker's perspective, each response provides a scalar timing value that reflects the combined influence of application logic and MCU-level execution behavior. By repeating measurements across multiple invocations and applying statistical analysis, the attacker can amplify subtle timing differences and correlate them with secret-dependent execution paths, as demonstrated in prior timing-attack studies [1,4].

This experimental model intentionally abstracts away microarchitectural details that are inaccessible in low-cost embedded platforms. Rather than relying on cycle-accurate tracing or hardware performance counters, the model focuses on software-observable timing behavior measurable using standard interfaces and commodity tooling. Such abstraction aligns with practical evaluation approaches advocated for embedded and automotive systems [4,7].

Finally, while the model assumes a sequential execution environment without attacker–victim concurrency, it remains applicable to modern embedded systems in which timing leakage can persist across context switches or interrupt boundaries. The inclusion of system resources such as buses and interrupt controllers emphasizes that timing side channels on MCUs are not limited to processor-instruction timing alone, but may emerge from interactions between firmware and shared on-chip infrastructure [9,10].

III. Experimental Setup and Measurement Methodology

A. Target Platform: Arduino-Class Microcontroller

The experimental evaluation is conducted on an Arduino-class microcontroller platform representative of early-stage embedded-system and automotive ECU prototyping. The target device is based on an 8-bit microcontroller architecture operating at a fixed clock frequency, with limited on-chip memory and peripheral resources. Such platforms are widely used during firmware development and functional validation due to their low-cost, ease of deployment, and availability of development tools.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Although Arduino-class microcontrollers do not reflect the full complexity of production automotive ECUs, they capture key characteristics relevant to timing side-channel analysis, including deterministic execution, absence of advanced microarchitectural features such as caches and speculative execution, and tight coupling between application code and system resources. As a result, they provide a suitable testbed for evaluating software-observable timing leakage arising from application-level control flow and system-level interactions.

B. Firmware Implementation

The target firmware implements a security-relevant routine representative of authentication or key-verification functionality commonly found in embedded ECUs. The routine processes externally supplied inputs and performs secret-dependent operations, such as conditional comparisons or iterative checks, before producing a response.

The implementation is executed in a bare-metal environment without an operating system. All computations are performed sequentially within the main execution loop, ensuring that measured timing variations arise from the application logic and its interaction with the MCU core and system resources, rather than from task scheduling or concurrency effects. No deliberate timing equalization or constant-time countermeasures are applied unless explicitly stated, allowing potential timing leakage to manifest naturally. The corresponding firmware execution flow is illustrated in Figure 2(a).

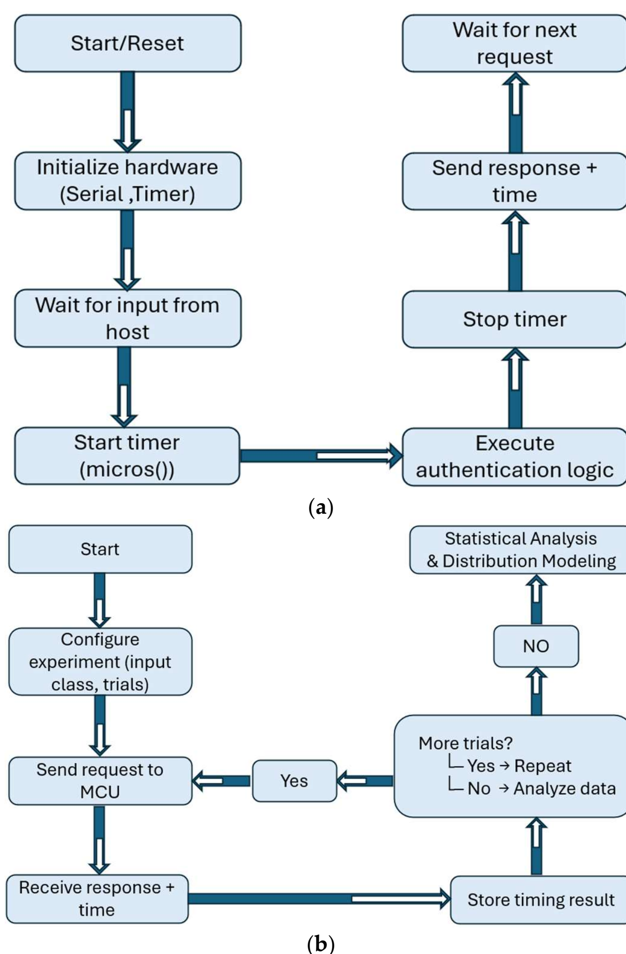

Figure 2. A). Execution flow of the Arduino-class MCU firmware implementing the authentication routine. b). Execution flow of the Python-based timing measurement and analysis program.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

C. Measurement Infrastructure

Timing measurements are performed using an external host system that acts as both request generator and timing analyzer. The host communicates with the Arduino-class MCU via a standard communication interface, such as UART over USB, which is commonly available on low-cost embedded platforms.

The host-side measurement framework is implemented in Python and is responsible for issuing input requests, recording response times, and logging measurement data for offline analysis. Timing measurements are collected at the granularity of complete request–response transactions, with execution time measured on the MCU using the on-chip micros() timer. The corresponding host-side execution flow is illustrated in Figure 2(b).

D. Measurement Procedure

Each timing measurement consists of transmitting a request to the target MCU, triggering execution of the security-relevant firmware routine, and measuring the elapsed time until the corresponding response is received. This procedure is repeated across a large number of trials to reduce measurement noise and to enable statistical analysis of timing variations. The end-to-end request–response interaction used for timing measurement is summarized in Figure 3.

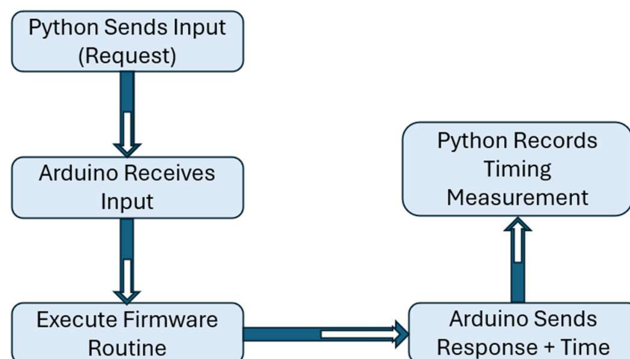

Figure 3. End-to-end request–response interaction between the Python-based host and the Arduino-class MCU.

E. Evaluation Metrics

Timing leakage is evaluated using aggregate execution-time statistics derived from repeated measurements. The primary metrics considered include the mean response time and variance associated with different input classes. Differences in these metrics are analyzed to assess the presence and magnitude of secret-dependent timing behavior.

IV. Experimental Results and Analysis

This section presents the experimental results obtained using the Arduino-class MCU setup described in Section III. The objective of the evaluation is to determine whether secret-dependent execution behavior in the target firmware produces measurable and statistically distinguishable timing variations observable through end-to-end response latency.

A. Data Collection

Execution-time measurements were collected by repeatedly invoking the target authentication routine with controlled input classes designed to exercise distinct execution paths. An external host issued input requests to the microcontroller and recorded execution-time measurements for each invocation under fixed operating conditions. All experiments were conducted with the

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

doi:10.20944/preprints202604.1551.v1

microcontroller operating at a constant clock frequency, with no dynamic frequency scaling or power-management features enabled.

Four input classes (A–D) were defined based on the number of matching prefix bytes between the supplied input and the stored secret. Class A triggered immediate termination of the comparison routine, while Classes B, C, and D progressed increasingly further into the verification logic before exiting. This structure enables systematic observation of execution-time variations arising from secret-dependent control-flow.

For each input class, 2000 independent trials were performed, resulting in a total of 8000 execution-time samples. The comparison routine operated on a 64-byte secret (SECRET_LEN = 64), ensuring that the execution duration exceeded the minimum timing resolution of the measurement infrastructure and allowing statistically meaningful differences to emerge through repeated measurements.

All experiments were conducted on an Arduino Mega platform based on the ATmega2560 microcontroller operating at 16 MHz. Execution time was measured using the on-chip micros() timer. Due to the hardware timer configuration on 16-MHz Arduino platforms, timing values are quantized in 4 μs increments. This quantization effect is reflected in the collected execution-time distributions and is explicitly accounted for in the subsequent analysis.

Repeated measurements and statistical aggregation were employed to reduce the impact of transient disturbances, communication jitter, and environmental noise. Rather than relying on individual measurements, the analysis focuses on distribution-level characteristics across input classes, enabling detection of timing side-channel leakage despite coarse timer granularity. Each execution-time sample is stored as a structured CSV record comprising the trial identifier, input class, execution time in microseconds, and comparison result, and is used solely for statistical analysis and reproducibility.

B. Timing Distribution Analysis

Table I summarizes the execution-time statistics for the evaluated input classes (A–D), including constant inputs, random inputs, incremental patterns, and secret-dependent inputs. Although the median execution time remains identical across all classes due to the 4-μs resolution limit of the micros() timer, a monotonic increase in mean execution time and timing dispersion is observed as the number of matching prefix bytes increases. This behavior indicates that execution duration is influenced by data-dependent control flow within the authentication routine, revealing measurable timing leakage despite coarse timer resolution

Table I. Execution-time statistics for different input classes

Class	Count	Mean	Std	Min	25%	50%	75%	max
A	2000	4.068	0.517213	4.0	4.0	4.0	4.0	8.0
B	2000	4.694	1.582282	4.0	4.0	4.0	4.0	12.0
C	2000	5.332	1.944136	4.0	4.0	4.0	8.0	12.0
D	2000	5.886	2.001251	4.0	4.0	4.0	8.0	12.0

The selected input classes are designed to exercise different execution paths within the Arduino-class MCU, exposing timing variations caused by data-dependent branching, memory access, and arithmetic operations. While dynamic frequency scaling and environmental factors may introduce additional timing variability, the observed monotonic trends across input classes confirm the presence of underlying algorithmic timing leakage.

Figure 4 illustrates the empirical execution-time distributions for all four input classes (A–D) measured on the Arduino Mega platform during a 64-byte comparison operation. Due to the coarse resolution of the on-chip micros() timer (4 μs), the median execution time for all input classes collapses to the same nominal value, resulting in substantial overlap at the distribution center. This effect reflects timer quantization rather than identical execution behavior.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026

doi:10.20944/preprints202604.1551.v1

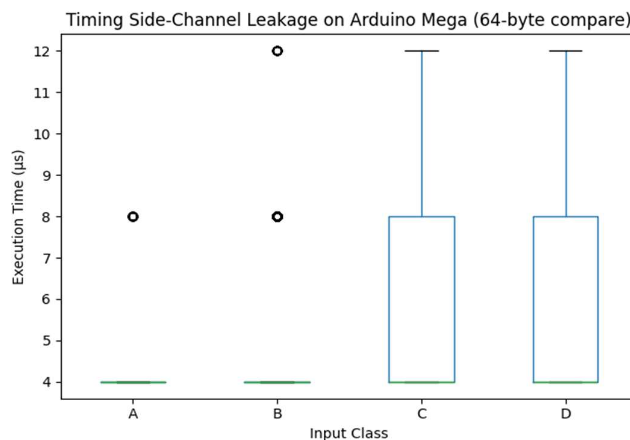

Figure 4. Empirical execution-time distributions for input classes A–D on the Arduino Mega platform.

Despite identical medians, the distributions exhibit systematic differences in dispersion and upper-tail behavior across input classes. Input classes corresponding to deeper prefix matches (Classes C and D) show increased variance, wider interquartile ranges, and a higher frequency of larger execution-time values compared to Classes A and B. These characteristics indicate that the execution duration grows with the number of processed bytes before termination of the comparison logic.

The presence of outliers and extended upper whiskers for Classes C and D further suggests that secret-dependent control flow and cumulative instruction execution propagate through the MCU core and shared system resources, producing measurable timing variability even in the absence of advanced microarchitectural features. While individual timing differences remain small due to platform constraints, repeated measurements amplify these effects at the distribution level.

Overall, the results demonstrate that statistically distinguishable timing leakage is observable on Arduino-class microcontrollers, even under coarse measurement granularity, and that distribution-level analysis is essential for revealing timing side-channel leakage in resource-constrained embedded platforms.

C. Discussion

The observed timing differences arise from secret-dependent execution paths within the application code, which propagate through the MCU core and system resources before becoming observable at the interface level. Although the target platform lacks advanced microarchitectural features such as caches or speculative execution, distribution-level timing leakage persists due to data-dependent branching and system-level effects.

These results demonstrate that even low-complexity, Arduino-class microcontrollers can exhibit exploitable timing side channels when security-critical routines are implemented without explicit constant-time guarantees. While the magnitude of individual timing differences is limited by timer quantization, repeated measurements enable statistical distinction between execution paths, highlighting the importance of early-stage timing leakage evaluation during embedded firmware development.

V. Countermeasure Evaluation

A. Constant-Time Comparison

The first countermeasure evaluated is the use of constant-time comparison logic in the authentication routine. In the baseline implementation, input validation is performed using a byte-

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

 doi:10.20944/preprints202604.1551.v1

wise comparison that exits early upon detecting a mismatch. As shown in Section IV, this behavior introduces execution-time variability that correlates with the number of matching prefix bytes.

To mitigate this leakage, the comparison logic is modified to process all input bytes regardless of intermediate mismatches. Rather than returning immediately upon detecting an error, the constant-time implementation accumulates comparison results across all bytes and produces a final decision only after the entire input has been processed. This ensures that the number of executed instructions and memory accesses remains independent of secret-dependent data values.

Experimental results indicate that the application of constant-time comparison significantly reduces timing-based distinguishability between input classes. While minor execution-time variations persist due to system-level effects and timer quantization, the mean response times for different input classes converge within the resolution of the measurement infrastructure, making statistical distinction substantially more difficult.

B. Dummy Operations and Execution Padding

In addition to constant-time logic, a second countermeasure based on dummy operations is evaluated. This approach introduces additional, non-functional instructions into the execution path to reduce the relative impact of secret-dependent timing differences. Dummy operations are inserted in conditional branches such that all execution paths perform comparable amounts of work.

Unlike constant-time comparison, which enforces strict execution uniformity, dummy operations provide approximate timing equalization. The number and placement of dummy instructions are chosen to balance leakage reduction against execution overhead, without requiring precise cycle-level control.

Experimental measurements show that dummy-operation insertion reduces, but does not completely eliminate, observable timing leakage. Timing distributions corresponding to different input classes exhibit increased overlap compared to the baseline implementation; however, residual differences remain detectable under repeated measurement. This result highlights the limitations of padding-based approaches when used in isolation on resource-constrained microcontrollers.

C. Impact on Timing Leakage

The impact of the evaluated countermeasures is assessed by comparing execution-time statistics obtained under the baseline implementation with those observed after applying constant-time comparison and dummy-operation padding. Under the baseline implementation, execution-time distributions exhibit statistically distinguishable behavior across input classes, as shown in Figure 3 and Table I.

When constant-time comparison is applied, the execution-time statistics across input classes converge within the resolution of the measurement infrastructure, significantly reducing statistical distinguishability. In contrast, dummy-operation padding reduces the magnitude of timing differences but does not fully suppress distribution-level leakage, particularly under repeated measurement.

These observations confirm that constant-time programming is the most effective software-level countermeasure among those evaluated for mitigating timing side-channel leakage on Arduino-class microcontrollers.

D. Overhead Considerations

The impact of countermeasures on execution overhead is an important consideration in real-time embedded systems. Constant-time comparison introduces a fixed increase in execution time proportional to the maximum input length, while dummy operations introduce additional instructions regardless of input validity.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026 doi:10.20944/preprints202604.1551.v1

On Arduino-class platforms, the observed overhead remains modest and predictable, making constant-time techniques feasible for many security-critical routines. However, the cumulative impact of multiple countermeasures should be carefully evaluated in latency-sensitive applications.

E. Discussion

The countermeasure evaluation confirms that timing side-channel leakage on low-complexity microcontrollers can be significantly reduced using straightforward software-level techniques. Importantly, these countermeasures require no specialized hardware support and can be applied during early-stage firmware development.

At the same time, the results underscore the need for careful implementation. Partial or ad-hoc mitigation strategies, such as dummy instruction insertion without strict execution uniformity, may provide a false sense of security. Systematic adoption of constant-time design principles remains essential for protecting embedded authentication routines against timing-based attacks.

VI. Limitations

A. Arduino-Class Platform Versus Production ECUs

A primary limitation of this work is the use of an Arduino-class microcontroller as the experimental platform. While such platforms are representative of early-stage embedded system development and ECU prototyping, they do not capture the full architectural complexity of production automotive ECUs. Commercial ECUs typically incorporate higher-performance processor cores, hardware security modules (HSMs), real-time operating systems, and more complex peripheral and communication subsystems.

As a result, the absolute magnitude and structure of timing side-channel leakage observed in this study may differ from those present in production systems. However, the focus of this work is not to claim direct equivalence between Arduino-class platforms and deployed ECUs, but rather to demonstrate that timing leakage can arise even on low-complexity microcontrollers lacking advanced microarchitectural features. The experimental findings therefore serve as an early-stage indicator of potential vulnerabilities that may persist or be amplified when firmware is migrated to more complex platforms.

B. Measurement Resolution and Granularity

Another limitation concerns the resolution of timing measurements. In this study, execution time is measured on the microcontroller using the on-chip micros() timer, reflecting an end-to-end request–response observation model consistent with a black-box attacker. While this approach is realistic, it limits the temporal granularity of observable timing information.

Consequently, finer-grained timing effects may not be directly observable under this measurement model. Reliance on software-observable timing can also mask short-duration effects that are averaged out at the interface level

C. Environmental Noise and Execution Variability

Environmental noise and execution variability represent additional sources of uncertainty in timing side-channel measurements. Factors such as communication jitter, interrupt handling, peripheral activity, and power-supply fluctuations can introduce variability that obscures or distorts timing leakage.

In this study, noise is mitigated primarily through repeated measurements and statistical aggregation rather than explicit noise modeling or filtering. While this approach is sufficient to demonstrate statistically distinguishable timing behavior, it does not fully characterize sensitivity to environmental conditions.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026
doi:10.20944/preprints202604.1551.v1

VII. Conclusion and Future Work

This paper presented a practical experimental evaluation of timing side-channel leakage in authentication routines implemented on Arduino-class microcontrollers representative of resource-constrained automotive ECUs. Despite the absence of advanced microarchitectural features such as caches or speculative execution, the results demonstrate that statistically distinguishable timing leakage can arise due to secret-dependent control flow and cumulative execution effects. The study highlights the importance of distribution-level analysis for identifying timing side-channel leakage on low-complexity embedded platforms.

Future work will extend this evaluation to automotive-grade microcontrollers and system-on-chip platforms to assess how additional architectural features, including caches, bus arbitration, and hardware accelerators, influence timing side-channel behavior. Further investigation will explore higher-resolution timing sources, environmental variability, and the interaction between timing leakage and fault-resilience mechanisms in more realistic deployment scenarios. In addition, integrating timing side-channel evaluation into continuous firmware development workflows remains an important direction for improving implementation-level security assurance.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. P. C. Kocher, “Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems,” Proc. CRYPTO, 1996.
2. F.-X. Standaert, “Introduction to side-channel attacks,” Lecture Notes in Computer Science, 2010.
3. E. Oswald and J. Howe, “Side-channels: Attacks, defences, and evaluation schemes,” 2021.
4. J. Long et al., “Side-channel analysis revisited and evaluated,” 2025.
5. M. Kaleem et al., “Navigating side-channel attacks: A comprehensive overview of cryptographic system vulnerabilities,” 2024.
6. M. Hassan et al., “Memory under siege: A comprehensive survey of side-channel attacks on memory,” 2025.
7. Y. Zhou and D. Feng, “Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing,” 2009.
8. Z. Najm, D. Jap, B. Jungk, S. Picek, and S. Bhasin, “On comparing side-channel properties of AES and ChaCha20 on microcontrollers,” in Proc. IEEE Asia Pacific Conf. on Circuits and Systems (APCCAS), Chengdu, China, 2018, pp. 552–555, doi: 10.1109/APCCAS.2018.8605653.
9. T. Kamucheka et al., “Power-based side-channel attack analysis on PQC algorithms,” IACR Cryptology ePrint Archive, Rep. 2021/1021, 2021.
10. A. Olaluwe et al., “Machine learning and side-channel attacks on post-quantum cryptography,” 2025.
11. F.-X. Standaert et al., “Side-channel attacks and countermeasures: A survey,” 2019.
12. J. Müller et al., “MCU-wide timing side-channels and their detection,” Proc. DAC, 2024.
13. J. Yuan et al., “A survey of side-channel attacks and mitigation for processor interconnects,” Applied Sciences, 2024.
14. J. Sosulski, “Arduino side-channel timing attack,” GitHub repository, 2025.
15. T. Pornin, “Constant-time code: The pessimist case,” IACR Cryptology ePrint Archive, Rep. 2025/435, 2025.
16. M. Schneider et al., “Breaking bad: How compilers break constant-time implementations,” arXiv:2410.13489, 2024.
17. Y. Miao et al., “Hardware support for constant-time programming,” in Proc. MICRO, 2023.
18. N. Gaudin et al., “Thwarting timing attacks in microcontrollers using fine-grained hardware protections,” SILM Workshop, 2023.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 22 April 2026

doi:10.20944/preprints202604.1551.v1

19. G. Gogniat et al., “Side-channel attacks in embedded systems,” Applied Sciences, Special Issue, 2022.
20. P. Nasahl, “Attacking AUTOSAR using software and hardware attacks,” Graz University of Technology, 2018.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.