Formally-Verified, Tight Timing Constraints for Machine Code

Charles Averill
charles@utdallas.edu
University of Texas at Dallas
Dallas, Texas, USA

Abstract

Timing excesses and leaks in binary code constitute critical threats to the availability and confidentiality of cyberphysical and cryptographic software systems. We introduce Cloq, the first dependently typed, machine-checked framework for verifying timing properties of raw (stripped) machine code within the Rocq interactive theorem-proving environment. By integrating formal timing models with sound symbolic execution, Cloq provides high-assurance, high-precision timing guarantees that verify that real-time systems and cryptographic algorithms meet their critical performance and security requirements. We demonstrate Cloq's generality with proofs of timing safety for the FreeRTOS kernel and cryptographic ciphers such as ChaCha20, and evaluate against a SOTA worst-case execution time tool. Validation on real hardware confirms accuracy and soundness of formal timing predictions.

1 Problem and Motivation

Machine-checked proofs of integrity, availability, and confidentiality properties offer the strongest attainable safety and security assurance for mission-critical software. But formal verification of timing-sensitive availability and confidentiality properties has remained relatively challenging and elusive for real-world production codes compared to integrity. This is in part because although mainstream software development tools (e.g., C compilers) strive for semantic transparency of data values, they make myriad code generation decisions that arbitrarily affect timing, including binary instruction selection, optimization, and reorganization. This frustrates accurate evaluation of timing properties through source code analysis alone; binary analysis is essential.

Timing decisions affect service availability when missed deadlines or excessive latency cause a system to fail or become unresponsive [10]. This threatens safety in hard real-time contexts where failure can be catastrophic, such as in critical closed-loop control applications for aerospace and power plant control systems. Confidentiality is affected when variations in execution time leak information about secret data, enabling timing side-channel attacks. Ensuring timing safety therefore requires reasoning about both performance and information flow through time-dependent behavior.

1.1 Real-Time Systems

Real-time systems underpin a vast array of mission-critical applications, including avionics, automotive control systems, industrial automation, and medical devices. The U.S. Federal Aviation Administration (FAA), Department of Transportation (DOT), and other regulating agencies therefore enforce strict *worst-case execution time* (WCET) requirements for software in these domains [12–14, 20, 21, 33] in an effort to ensure that control loops and other real-time functions execute within precise time intervals to maintain system stability and correctness. Failure to meet these timing constraints can result in catastrophic failures, such as aircraft control loss or medical device malfunctions [36].

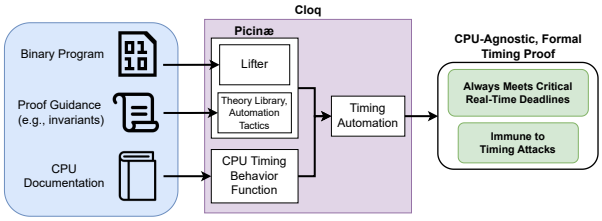


Figure 1: Cloq Pipeline

Specialized hardware is typically employed to meet WCET standards. Suitable microprocessors, such as the PowerPC e200, ARM Cortex-R, and LEON3, must support stable, formalizable timing guarantees with cycle-accurate timings [6, 9, 18].

Despite these precautions, traditional WCET analysis techniques, such as control flow analysis and measurement-based execution time analysis [37], supply only anecdotal evidence of compliance. They forgo formal guarantees in favor of automation, allowing large software systems to be rapidly tested, but without exhaustive coverage of the input space or precise timing dependency recovery.

1.2 Constant-Time Cryptography

Cryptographic implementations present a different but related challenge. Timing side-channel attacks exploit variations in execution time to infer secrets, breaking confidentiality even when cryptographic algorithms are mathematically sound.

Classic attacks, such as Kocher's timing attacks on RSA [24], demonstrate that even minute timing differences in modular arithmetic operations can leak secret key bits. Recent work has extended these attacks to cache-based timing attacks [2, 31] and branch prediction or speculative execution attacks, such as Spectre and Meltdown [23]. Defenses against timing attacks require ensuring that execution time remains independent of secret data, but achieving this property in practice is difficult due to microarchitectural effects that are often poorly documented and hardware-specific.

1.3 Motivation

Traditional verification techniques focused on bug-finding are insufficient for obtaining airtight, machine-checkable formal guarantees about timing properties. For example, real-time verification methods rely on conservative WCET bounds that often do not accurately reflect true execution behavior [5, 22, 25, 38]. Cryptographic verification techniques, such as constant-time programming methodologies [29], require careful manual implementations that do not prove the absence of timing leaks. Furthermore, these methods are typically applied at the source level, which does not always guarantee that the *binary* code being run is actually free of timing vulnerabilities [3, 8, 19, 26, 34, 39]. The dearth of comprehensive formal methods for timing verification of binaries leaves many

PLDI SRC '25, June 16–20, 2025, Seoul, SK
© 2025 Association for Computing Machinery.

safety-critical and security-critical systems vulnerable, calling into question their reliability and trustworthiness.

Addressing this problem requires new formal verification techniques capable of rigorous mathematical reasoning about execution times of binary native codes. Such techniques must account for hardware-level execution behaviors while remaining applicable to real-world software development workflows. The development of precise, automated proof techniques for timing correctness will both improve safety in real-time systems and enhance security in cryptographic implementations, ensuring that these systems can be trusted even in adversarial environments.

2 Background and Related Work

2.1 Bottom-up Formal Methods

Our work builds upon Picinæ [1, 17], a framework within the Rocq interactive theorem prover (ITP) for the development of functional correctness proofs for arbitrary (e.g., non-compiled) machine code.

2.1.1 Lifting. Picinæ uses a low-level intermediate language (IL) formalized within the Rocq proof assistant that is similar to other ISA-modeling ILs [4, 28] but dependently-typed and strongly normalizing to comply with Rocq's foundations in the calculus of inductive constructions. It represents machine instructions as structured, effect-preserving transformations of an abstract state. Programs are lifted to a partial map from virtual addresses a to IL that encodes the operational effect of the instruction at a on an abstract cpu state. IL effects include state variable-assignments, control transfers, and conditionals. The genericity of the IL makes lifting achievable for a wide class of ISAs, making Picinæ source language-agnostic.

Expressions in the IL comprise state element reads, memory operations, and modular arithmetic. Memory is modeled as a pure vector-valued IL variable. Unreliable, implementation-defined ISA behaviors are modeled as non-deterministic assignments in the IL, preventing proofs from relying on hardware idiosyncrasies that might be artifacts of manufacturing anomalies or defects.

2.1.2 Invariants. To facilitate formal reasoning about lifted code, Picinæ introduces *invariant sets*, which define untrusted, machine-verified properties asserted at specific code points. This forms a basis for co-inductively proving security and correctness guarantees. They are implemented as a partial map from virtual addresses to cpu state propositions, where propositions may range over Rocq's full higher-order, dependent propositional specification language.

2.1.3 Symbolic Execution. Picinæ includes a verified symbolic interpreter to analyze lifted machine code within a Rocq proof context. This interpreter enables stepwise execution of an abstract machine state, incorporating Rocq proof meta-variables where necessary to model unknowns. It leverages dependent typing to automatically attach ISA-specific properties to untyped binary state elements within each proof context. For example, w-bit register values have Σ-type $\{n : \mathbb{N} \mid n < 2^w\}$. This affords machine-checked proofs of code properties that rely on ISA-specific properties.

Because the interpreter introduces proof goals corresponding to all possible cases of each branch, complete code coverage is guaranteed—any coverage lapse yields a proof goal that must be solved by contradiction to prove that the code site is unreachable at runtime. Invariant sets thereby prove coverage completeness.

2.1.4 Traces. Program traces in Picinæ are constructed by following execution paths within the lifted IL representation. They capture the sequence of state transitions induced by instruction execution, providing a formal basis for reasoning about control flow and program behavior. By integrating trace analysis with invariant reasoning, Picinæ facilitates proofs of temporal correctness, security, and reachability properties expressible in LTL [32].

2.2 Existing Formal Timing Approaches

2.2.1 Abstract Interpretation statically approximates program behavior by interpreting it with abstract values. This can determine an upper bound for execution times using control-flow analysis, where paths through the program are modeled as a set of constraints.

2.2.2 Measurement-Based Analysis involves executing the code on bare hardware or a simulator, and measuring the execution time for various input sets, recording the maximum execution time observed. Although this can more easily produce statistical results for complex systems, it is not a comprehensive search over code paths and offers no formal guarantees. The method's precision can be improved by collecting more measurements, including varying the initial processor state or by analyzing multiple test cases.

3 Approach and Novelty

Our approach defines ISA *timing modules* in Rocq, which model each cpu's timing behavior to provide machine-checkable reasoning about timing properties. It provides high-assurance timing guarantees for machine code through a rigorous pipeline that prevents false assurances and is approachable to a wider user base than standard formal verification tasks. We here present a RISC-V instantiation of Cloq, but we have also developed instantiations for x86-64, ARMv6, ARMv7, and MSP430.

3.1 Instruction Timing

3.1.1 Units. We select cpu cycle counts as our unit of time because it is the standard for most WCET regulations and constitutes a granular, consistent measure of execution time that is translatable to clock time. This affords precise analysis of performance and resource utilization, ensuring predictable behavior in timing-sensitive systems. Cycle counts also facilitate comparison and optimization across different processors and hardware configurations.

3.1.2 CPU Selection. The NEORV32 RISC-V cpu is a high-reliability microprocessor designed for timing-sensitive computations. Its timing behavior is documented as a detailed datasheet [30] that emphasizes analysis-amenable properties, such as non-speculative execution. In this work we do not support out-of-order execution, since our focus is on areas such as flight control, where manufacturers choose cpus with better timing predictability [9, 18].

3.1.3 Implementation. We encode instruction timings as a Rocq function that maps a machine instruction's type, arguments, and additional parameters (e.g., memory latency), to its cycle count. The cycle count computation takes into account special instructions such as CLZ (count leading zeros) and shift operations, whose latencies depend on the immediate value or register values.

Instruction latency may also depend on the instruction history. Manufacturer-provided instruction WCET documentation expresses such timings as formulas over relevant cpu state elements, which are incorporated into our Rocq timing function. Our approach thereby yields generalized timing guarantees expressed as a formula whose parameters can include hardware-specific conditions and tolerances. Such a formula reveals how the parameters must be constrained to achieve desired goals, such as worst-case bounds or zero information leakage.

```
add:
    beqz   t0, end     ; 0  - goto end if t0 == 0
    addi   t1, t1, 1   ; 4  - increment t1
    addi   t0, t0, -1  ; 8  - decrement t0
    j      add         ; 12 - goto add
end:                   ; 16
```

Figure 2: Peano addition assembly code implementation

```
Definition timing_invs (p:addr) (x y:N) (t:trace) :=
 let tb := 5+(ML-1) in   (* time of a taken branch *)
 let ft := 3 in (* time of a fallen-through branch *)
 match t with (Addr a, s) :: t' ⇒ match a with
 | 0 ⇒ Some (s R_T0 ≤ x ∧
        cycle_count t' = (x - T0) * (ft + 2 + 2 + tb))
 | 16 ⇒ Some (cycle_count t' = tb + x * (ft + 2 + 2 + tb))
 | _ ⇒ None end | _ ⇒ None end.
```

Figure 3: Invariant set for the addloop code in Fig. 2

3.2 Trace Timing

Extending Picinæ's symbolic interpreter to model timing properties entails mapping the instruction timing function onto the cpu trace, yielding a list of cycle counts. This list is then summed, providing the total number of cycles taken to reach the exit point of a function starting from an entry point. Timing properties universally quantify over traces, expressing properties of all possible executions.

3.3 Proof Structure

Timing proofs tend to be considerably more amenable to proof automation than full functional correctness proofs. Figure 2 illustrates via an example loop that implements Peano addition, and Figure 3 shows a suitable invariant set for the code, consisting of a precondition, loop invariant, and postcondition.

The loop invariant characterizes the loop's timing behavior and tracks critical information for loop termination. It proposes that the cycle count up to the loop's current iteration is equal to $(c_0 - c)t$, where c_0, c, and t are the initial loop counter, the current loop counter, and the time of the loop body, respectively. The postcondition asserts that the total time taken is equal to $t_0 + c_0 t$.

The structure of the invariant set directly follows from the control flow graph (CFG) of `addloop`. The proof's structure is isomorphic to the invariant set's structure, requiring only standard machinery of Rocq's proof system and Picinæ 's automatic binary arithmetic simplifier, and follows directly from the CFG.

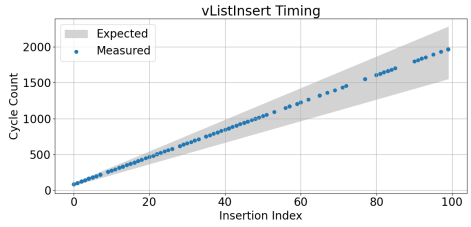
Picinæ's abstract interpreter provides a `step` tactic that advances the cpu state by one instruction. The core of most timing proofs consists of stepping forward until an invariant is reached (`repeat step`), auto-simplifying the binary arithmetic expressions accumulated during the steps (`psimpl`), and then generating a proof by reflexivity of the invariant-defined timing expression's equality to the proof-generated timing expression, often via Rocq's solver for linear integer arithmetic (`lia`). All three of these steps are largely automated—we provide a `whammer` tactic that performs these actions automatically, as well as a lower-level `hammer` tactic that makes fewer assumptions about the goal. The shared structure of many timing proofs allow our automation tactics to carve a straightforward path toward high-assurance timing proofs for many codes.

4 Results and Contributions

Cloq allows software developers to obtain high-assurance timing guarantees for mission-critical machine code. Its outputs are easily interpreted by developers familiar with assembly language, and proofs are often easy to develop even by users with limited knowledge of formal verification. To demonstrate, we present examples of real-world code with timing proofs evaluated on NEORV32.

```
Theorem addloop_timing:
  ∀ s p t, satisfies_all
    lifted_addloop                    (* lifted code *)
    (timing_invs p (s R_T0) (s R_T1))  (* invariants *)
    addloop_exit                      (* exit point *)
    t.                              (* abstract trace *)
Proof.
    (* Address 4 *) repeat step; psimpl; subst; lia.
    (* Address 8 (break/loop cases) *) whammer.
    (* Postcondition *) whammer.
Qed.
```

Figure 4: Abbreviated proof of Fig. 3 timing properties



Function	Expected	Measured	Trials
vListInitialise	29–42	34	200
vListInitialiseItem	9–14	10	200
vListInsertEnd	43–64	54	200
uxListRemove	48–72	59	125

Figure 5: Experimental Results for FreeRTOS List Functions

4.1 FreeRTOS

To evaluate Cloq on realistic embedded software, we formally verify the timing behavior of core FreeRTOS kernel functions for task management and scheduling. FreeRTOS is a widely used real-time OS for embedded systems [16], making it a realistic target for demonstrating the practical utility of machine-level timing proofs. We focus on two key subsystems: the task scheduler (`task.c`) and kernel linked lists (`list.c`), both of which are fundamental to kernel operation and time-critical code paths. Each verified function runs on RISC-V hardware using an expert-derived timing model, and yields an exact, machine-checked execution time.

4.1.1 task.c We formally verify the exact timing behavior of 4 functions from FreeRTOS's `task.c` module, which handles task creation, deletion, suspension, and transition:

- `vTaskSwitchContext` switches context to the highest-priority ready task and updates the task context.
- `prvResetNextTaskUnblockTime` updates the kernel's earliest unblock time for a blocked task.
- `vTaskSuspendAll` suspends task scheduling to allow kernel operations to proceed atomically.
- `xTaskGetCurrentTaskHandle` returns a pointer to the task control block of the currently running task.

Of these, `vTaskSwitchContext` has highest importance and complexity, running between high-priority tasks with timing critical to system operation, and using non-terminating loops to safely handle stack overflows. Its proof was developed concurrently with the Cloq infrastructure and took one expert a month to complete.

Table 1: Comparison of Cloq with WCET predictions (WCP) and observations (WCO)

Program	RapiTime WCP	RapiTime WCO	Cloq Prediction	Improvement Factor*
`sum` ($n \leq 1000$)	436,933,677	469,356	$(n = 0)\ ?\ 38 : (50 + 18n)$	2909×
`while_true_break` ($n \leq 1000$)	87,811,117	114,234	$(n \leq 2)\ ?\ 52 : (4 + 26n)$	3504×
`arraysearch` (idx, $len \leq 1000$)	65,244,002	81,071	$76 + 26\min(idx, len)$	16759×
`list_find` (idx, $len \leq 128$)	10,824,648	88,333	$94 + 99\min(idx, len)$	4721×
`list_insert_at_n` (idx, $len \leq 128$)	1,446,238	16,478	$50 + 31\min(idx, len)$	2709×
`sorted_list_insert` ($idx \leq 130$)	250,768	5,901	$49 + 24idx$	356×
`collatz`	✘	✘	3,355,098	∞

*Improvement factor is $\frac{1}{m}\sum_{n=0}^{m}(WCP/f(n))$ where m is the WCP max input size and f is the Cloq prediction.

4.1.2 list.c We formally verify the timing behavior of all functions in FreeRTOS's `list.c` module, which handles the creation and modification of linked lists for the kernel:

- `vListInitialise` initializes a list structure, zeroing its item count and appending a sentry node at the end.
- `vListInitialiseItem` prepares an item for insertion.
- `vListInsertEnd` inserts a list item at the end of a list, just before the sentry node, preserving insertion order.
- `vListInsert` inserts an item, preserving sortedness.
- `uxListRemove` removes a list item, updating the item's container pointer and the list's item count.

FreeRTOS uses doubly-linked, cyclic, non-empty linked lists, which have implementations significantly more complex than standard linked lists. `vListInsert` is our foremost example of these challenges, as it includes a loop with a complex termination condition. The loop searches for the sorted position of the new element, but proving that it finds one in the presence of cycles requires proving and reasoning about the uniqueness of nodes in a cycle.

Because the scheduler uses lists to store task information, precise timing of this function is critical to the safe operation of real-time systems. Two expert users completed the single-invariant proof for `vListInsert` in 15 hours, while one expert user completed the proof for the remaining list functions in 1 hour.

4.2 Constant-Time Cryptography

We formally verify the absence of timing side-channel vulnerabilities in an implementation of ChaCha20-Block developed for RISC-V [11]. This operation is core to the ChaCha20 cipher [29]. Our proof establishes that its timing is not dependent on program variables, and its loop has a constant iteration bound.

The absence of data dependencies in the verified post-condition proves that it has no cycle-level timing leaks of any sensitive data. The predicted execution time of 12375–14849 cycles matches empirical measurements that exhibit a consistent 13624 cycle count.

The proof script is roughly 80 lines of Rocq and requires one loop invariant, completed by one expert in 2 hours. Cloq's capability to generalize instruction semantic effects keeps the proof small relative to the code complexity. The many arithmetic and memory operations are generalized by Cloq's machine-verified proof tactics for abstracting irrelevant processor state elements, avoiding state space explosion. These generalize effects that do not affect control-flow, limiting the state space while retaining critical timing data.

We additionally formally verify the timing safety of `ct-swap`, an important cryptographic primitive compiled for RISC-V. The function swaps the contents of two arrays on the condition of a confidential function argument, and the underlying swap operation is implemented in OpenSSL [27]. Despite being algorithmically constant time, `ct-swap` is vulnerable to timing attacks on architectures with data-memory dependent prefetchers (DMP), such as the Apple-M series [7]. Our proof shows that on a processor without a DMP, the timing behavior is not parameterized by the secret, preventing attackers from discerning secret array swaps from cycle count observations. The proof is roughly 50 lines of Rocq with one loop invariant, and was completed by an expert in 20 minutes.

4.3 Additional Experiments

We verified the timing behavior of several fundamental algorithms, including basic operations on simple data structures such as searches and sorts. This effort involved developing a general-purpose linked list theory module that facilitates proofs about linked list data structures. It supports arbitrary propositions over nodes in lists, preservation of a list between two different program memories, properties of well-founded lists, value and node membership, distance between nodes, and list sortedness. The module can be specialized to lists with varying value and pointer sizes and underlying memory bitwidth and endianness using Rocq's functor system.

4.4 WCET Comparison

To evaluate Cloq's precision relative to state-of-the-art (informal) WCET analysis, we analyze a set of hand-written C programs using Rapitime [35], one of the top commercial WCET products used for DO-178 and ISO 26262 certification. Each program is compiled to an x86-64 binary targeting an Intel i5-7300U processor. We construct a corresponding Cloq timing model using instruction latency data derived from published microarchitectural benchmarks [15]. We prove timing properties for each binary, and compare the resulting proof-derived execution times against the WCET estimates.

Table 1 summarizes the evaluation results. Cloq consistently produces tighter and more general timing bounds across all benchmarks by reporting closed-form expressions that capture how execution time scales with input values and sizes. As shown in the Improvement Factor column, Cloq's proof-derived timing bounds are significantly tighter than the static WCET predictions and closer to real, observed behavior. WCET's worst case observations (WCO) are bloated in part by the WCET instrumentation, whereas Cloq's analysis reflects real, unaltered compilations of the target code.

WCET's inability to analyze the `collatz` test shows its general susceptibility to code coverage errors. The benchmark applies the Collatz transform ($\lambda n\,.\,(even\ n)\ ?\ n/2 : 3n{+}1$) to its 16-bit unsigned integer input n for at most 55,000 iterations or until n reaches 1. If $n \neq 1$, it performs a long busy-wait before returning. However, since 55,000 is a strict upper bound on the number of iterations required for convergence on any 16-bit input, the busy-wait is unreachable in practice. Since the busy-wait does not affect the state of the system (such as the return value or I/O streams), gcc elides it during compilation. Static WCET analysis cannot reliably decide code reachability (reducible to the halting problem in the general case), and because it often relies on source-level analysis (missing

dead code elimination during compilation), it crashes trying to compute the resulting state and associated timing costs.

In contrast, the Cloq proof succeeds by generating four cases: (1) function entry to the first loop, (2) one loop iteration, (3) exiting the loop because bound has been reached, and (4) exiting the loop because the collatz value is 1. Because Cloq operates on machine code, it disregards the busy-wait loop that is optimized away. To prove the invariant at the end of each of these cases, the analysis rules out all but a small segment of the code and analyzes its effects on the abstract processor state after an arbitrary number of loop iterations. An expert user completed the proof in approximately 3 hours with 40 lines of proof code, while the WCET analysis crashes after 4 hours.

5 Conclusion

We presented Cloq, a framework for machine-checked proofs of timing behavior for native binary code. Cloq facilitates formal proofs of availability and confidentiality timing by filling gaps in traditional WCET analyses and CTC. Our RISC-V implementation proves timing of NEORV32 binaries, yielding machine-checked proofs of exact cycle counts of critical systems, including the FreeRTOS kernel's list and task modules. We also verify timing safety and leak-freedom of cryptographic software, including ChaCha20. Across all case studies, Cloq achieves cycle-count predictions that match hardware measurements down to exact instruction counts, and consistently outperforms commercial WCET tools by orders of magnitude.

References

[1] Charles Averill, Ilan Buzzetti, Alex Bellon, and Kevin W Hamlen. [n.d.]. LAPSE: Automatic, Formal Fault-Tolerant Correctness Proofs for Native Code. ([n. d.]).

[2] Daniel J. Bernstein. 2005. *Cache-timing Attacks on AES.* Technical Report. The University of Illinois at Chicago. cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[3] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. 2020. JIT Leaks: Inducing Timing Side Channels Through Just-in-Time Compilation. In *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P).* 1207–1222.

[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV).* 463–469.

[5] Hugues Cassé, Haluk Ozaktas, and Christine Rochange. 2015. A Framework to Quantify the Overestimations of Static Wcet Analysis. In *Proceedings of the 15th International Workshop on Worst-case Execution Time Analysis (WCET).* 1–10.

[6] Francisco J. Cazorla. 2013. Timing Properties of the LEON3-based GR712RC Board. Implications on Task Scheduling. Presented the Data Systems Division (TEC-ED) and Software Systems Division (TEC-ED) of the European Space Research & Technology Centre (ESTEC). https://indico.esa.int/event/42/contributions/2444.

[7] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *Proceedings of the 33rd USENIX Security Symposium.* 1117–1134.

[8] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P).* 1115–1132.

[9] Advanced Micro Devices. [n.d.]. Am29000 and Am29005 Streamlined Instruction Microprocessors. https://datasheets.chipdb.org/AMD/29K/00x_ds.pdf.

[10] Jakob Engblom, Andreas Ermedahl, Jan Mikael Sjödin, Gustafsson, and Hans Hansson. 2003. Worst-case Execution-time Analysis for Embedded Real-time Systems. *International Journal on Software Tools for Technology Transfer* 4, 4 (2003), 437–455.

[11] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. In *Proceedings of the 45th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI).* 1704–1729.

[12] FAA. 2022. AC 20-152A - Development Assurance for Airborne Electronic Hardware. https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1041323.

[13] FAA. 2024. 14 CFR 25.1309 – Equipment, systems, and installations. https://www.ecfr.gov/current/title-14/chapter-I/subchapter-C/part-25/subpart-F/subject-group-ECFR9f24bf451b0d2b1/section-25.1309.

[14] FAA. 2024. AC 20-193 - Use of Multi-Core Processors. https://www.faa.gov/regulations_policies/advisory_circulars/index.cfm/go/document.information/documentID/1036408.

[15] Agner Fog. 2025. *Instruction tables.* https://www.agner.org/optimize/instruction_tables.pdf

[16] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. 2016. Open source FreeRTOS as a case study in real-time operating system evolution. *Journal of Systems and Software* 118 (2016), 19–35.

[17] Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. 2019. Source-free Machine-checked Validation of Native Code in Coq. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST).* 25–30.

[18] Honeywell. [n.d.]. Pegasus FMS for Airbus A330 A320 Technical summary. https://aerospace.honeywell.com/content/dam/aerobt/en/documents/learn/platforms/brochures/C61-1647-000-000_ATR_TechnicalSummary_PegasusFMS_Airbus_A330_A320.pdf

[19] Intel. 2022. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html

[20] ISO. 2010. IEC 62304:2006. www.iso.org/standard/38421.html.

[21] ISO. 2018. ISO 26262-1:2018. www.iso.org/standard/68383.html

[22] Raimund Kirner and Peter Puschner. 2008. Obstacles in Worst-case Execution Time Analysis. In *Proceedings of the 11th IEEE International Symposium on Object and Component-oriented Real-time Distributed Computing (ISORC).* 333–339.

[23] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre Attacks: Exploiting Speculative Execution. *Communications of the ACM (CACM)* 63, 7 (2020), 93–101.

[24] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO).* 104–113.

[25] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A Survey on Static Cache Analysis for Real-Time Systems. *Leibniz Transactions on Embedded Systems (LITES)* 3, 1 (2016).

[26] Pedro Malagón, Juan-Mariano De Goyeneche, Marina Zapater, José M. Moya, and Zorana Banković. 2012. Compiler Optimizations as a Countermeasure against Side-Channel Analysis in MSP430-Based Devices. *Sensors* 12, 6 (2012), 7994–8012.

[27] Marius Marian and Eugen Sendroiu. 2009. A PKI case study: implementing the server-based certificate validation protocol. In *Proceedings of the 7th WSEAS International Conference on Information Security and Privacy (ISP).* 54–60.

[28] National Security Agency. 2017. *P-Code Reference Manual.* spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html.

[29] Yoav Nir and Adam Langley. 2015. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539.

[30] Stephan T. Nolting. 2025. The NEORV32 RISC-V Processor - Datasheet. stnolting.github.io/neorv32.

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the the Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA).* 1–20.

[32] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS).* 46–57.

[33] RTCA. 2011. *DO-178C: Software Considerations in Airborne Systems & Equipment Certification.* Technical Report. RTCA, Washington, DC.

[34] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. 2025. Breaking Bad: How Compilers Break Constant-Time Implementations. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 1690–1706.

[35] Rapita Systems. 2025. RapiTime. https://www.rapitasystems.com/products/rapitime

[36] DOLORES WALLACE and D. Kuhn. 2002. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability, Quality and Safety Engineering* 08 (07 2002). https://doi.org/10.1142/S021853930100058X

[37] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008).

[38] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenstrom. 2008. The Worst-case Execution-time problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.

[39] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *Proceedings of the 32nd USENIX Security Symposium.* 3655–3672.