

Tickling x86_64: Detecting Emulator Inaccuracies Through CPU Instruction Forensics

Lafosse Louis
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Splaiul Independenței 313, Bucharest, Romania, 060042
lafosse.louis@stud.acs.upb.ro
<https://github.com/louislafosse/research-polytechnic>

February 7, 2026

Abstract

CPU emulators are widely used in security infrastructure, from malware sandboxes to cross-architecture exploit development. Despite this, their accuracy on instruction corner cases has not been systematically evaluated. In fact, this research presents a structured assessment of x86-64 emulator behavior for security-relevant CPU instructions, testing eight commonly deployed emulators. To this end, we developed a trait-based testing framework in Rust that provides a uniform interface for executing test cases across different emulator implementations. The framework separates test logic from emulator-specific integration details, thereby enabling direct comparison of behaviors from native hardware execution to subprocess invocation and FFI bindings. Moreover, our tests revealed substantial inaccuracies in how emulators handle edge cases in floating-point operations, CPU flags, and processor identification instructions. Nevertheless, the extensible framework supports future research on additional CPU behaviors and architectures beyond x86-64, offering practical guidance for security researchers and emulator developers.

1 Introduction

1.1 Motivation

CPU emulators are widely used in modern computing, enabling cross-architecture execution, security research, and binary analysis. In particular, they power malware sandboxes, enable ARM devices to run x86 applications, and provide controlled environments for vulnerability research. However, emulators implement complex CPU behaviors through approximation, thereby potentially introducing behavioral discrepancies that have security implications.

Current emulator testing focuses primarily on functional correctness, checking whether programs execute without crashing. Nevertheless, this approach misses corner cases, the

subtle architectural behaviors that occur under unusual conditions. These boundary conditions matter for security research, where exploits often rely on precise CPU behavior [1]. Consequently, emulator mismatches cause false negatives in security testing, where malicious behavior goes undetected because the emulated environment fails to trigger the same conditions as native hardware. Moreover, these discrepancies enable anti-emulation techniques, fingerprinting, and sandbox escaping. The CVE-2021-44078 vulnerability exemplifies this risk, where a memory management flaw in Unicorn Engine allowed sandbox escape [4].

When emulators don't accurately model corner cases, security researchers may miss vulnerabilities or develop exploits that fail on real hardware. In reality, modern malware actively exploits behavioral differences between emulators and hardware to detect analysis environments and alter behavior accordingly. Additionally, beyond behavioral inaccuracies, emulators themselves are attack surfaces.

The 2015 ESET vulnerability demonstrated how implementation flaws in an antivirus emulator allowed remote root compromise through crafted files [5], showing that security products relying on emulation can become vectors for attack. Even legitimate applications depending on specific CPU features may malfunction silently in emulated environments, leading to false test results that undermine the entire purpose of testing in emulation.

1.2 Research Problem

Despite widespread emulator usage, no comprehensive evaluation exists for security-relevant corner cases. In particular, the security research community lacks detailed benchmarks comparing emulator accuracy on boundary conditions, proper documentation of emulator-specific limitations and bugs, and evidence-based guidelines for selecting appropriate emulators for security work. There is no established methodology for discovering vulnerabilities in the emulation frameworks themselves.

1.3 Contributions

This research presents a test suite for x86-64 emulator corner cases, focusing on security-relevant behaviors. Through comparative analysis of eight widely-used emulators across three fundamental instruction patterns, we discovered previously unknown bugs including unhandled flags in FPU stack overflow, Blink's auxiliary flag error and an architecture mismatch vulnerability in Unicorn Engine. Our extensible framework enables future research on additional CPU boundary behaviors while providing practical, evidence-based guidance for emulator selection in security research contexts.

1.4 Report Structure

The remainder of this report provides background on CPU architecture fundamentals and emulation techniques, surveys existing emulators and testing approaches, and identifies current gaps and research opportunities. We then describe our framework design and the technical challenges encountered during implementation, before presenting our experimental findings and their security implications. The report concludes with a discussion of future work and broader impact.

2 Background and Concepts

2.1 x86-64 Architecture Fundamentals

The x86-64 architecture (also called AMD64 or Intel 64) extends the 32-bit x86 instruction set with 64-bit addressing and additional features. Three specific aspects matter for this research:

2.1.1 x87 Floating Point Unit (FPU)

The x87 FPU provides hardware floating-point arithmetic using an 8-register stack architecture, with registers labeled ST(0) through ST(7). Unlike general-purpose registers, the FPU operates in a stack-based manner where operations push and pop values on a circular stack. Specifically, the 16-bit FPU Status Word register tracks the unit's state, including critical flags such as the Invalid Exception bit (IE, bit 0) indicating invalid arithmetic operations, the Stack Fault bit (SF, bit 6) detecting stack overflow or underflow, condition codes C0-C3 for comparison results, and the TOP field (bits 11-13) pointing to the current top-of-stack position.

When a program attempts to push a ninth value onto the 8-register stack, the hardware sets both the IE and SF flags to signal the overflow condition. Importantly, this behavior has security implications. The CVE-2018-3665 [3] vulnerability, known as "Lazy FPU State Restore," exploited FPU state handling for side-channel attacks, demonstrating that precise FPU behavior modeling is a security requirement, not merely an implementation detail.

2.1.2 EFLAGS Register and LAHF Instruction

The EFLAGS register contains CPU status and control flags. The LAHF (Load AH with Flags) instruction copies the low 8 bits into the AH register, thereby preserving flags in a format compatible with older 8086 processors. The bit layout includes the Sign Flag (bit 7), Zero Flag (bit 6), Auxiliary Carry flag (bit 4), Parity Flag (bit 2), and Carry Flag (bit 0), with reserved bits at positions 5, 3, and 1.

The Auxiliary Flag needs particular attention as it indicates carry or borrow between bits 3 and 4, primarily used for Binary Coded Decimal (BCD) arithmetic. While BCD operations are rare in modern software, incorrect AF behavior can break legacy BCD code or enable emulator fingerprinting, where malware detects it's running in an emulated environment by checking for known discrepancies in flag handling.

2.1.3 RDTSCP and Time Stamp Counter

The RDTSCP instruction (Read Time-Stamp Counter and Processor ID) returns both a 64-bit cycle counter in the EDX:EAX register pair and a processor identifier in ECX from the TSC_AUX model-specific register (MSR 0xC0000103). The monotonically increasing timestamp enables precise performance measurement, while the TSC_AUX value identifies which CPU core is executing, thereby supporting NUMA node affinity and thread-specific data structures. In fact, this processor identification capability makes RDTSCP particularly relevant for security research, as emulators that fail to implement TSC_AUX properly can be trivially fingerprinted.

2.2 Emulation Techniques

Modern CPU emulators employ several fundamental approaches to translate guest instructions to host execution. For instance, binary translation, exemplified by QEMU’s Tiny Code Generator (TCG), translates guest instructions to an intermediate representation before generating host-native code. This approach offers broad cross-architecture support and reasonable performance, though the complexity of modeling every architectural detail makes perfect emulation practically impossible.

Hardware-assisted virtualization leverages CPU features like Intel VT-x and AMD-V to execute guest code directly under hypervisor supervision. While this achieves near-native performance, it requires matching architectures and provides limited control over CPU state compared to software emulation. Meanwhile, Just-In-Time compilation is a hybrid approach, compiling frequently-executed code paths to native code for performance while interpreting cold paths, though this introduces startup overhead and implementation complexity.

Each technique involves fundamental trade-offs between performance, accuracy, and implementation complexity, with security implications arising primarily from the approximations necessary to make emulation practical.

3 State of the Art

3.1 Existing Emulators

The emulator landscape spans diverse implementations targeting different use cases. In particular, QEMU is the de facto standard for system emulation, employing its Tiny Code Generator for binary translation. Its user-mode variant allows running single binaries without full system emulation, thereby making it popular in cross-compilation workflows and security research despite its complex codebase and performance overhead.

Blink, developed by Justine Tunney, takes a different approach by focusing on fast x86-64 binary translation for running Linux, Windows, and macOS binaries with emphasis on portability and simplicity. However, it’s limited to x86-64 hosts. Meanwhile, Box64 targets a specialized niche, enabling x86-64 binaries on ARM64 systems like Raspberry Pi and Apple Silicon, which is particularly important for running games and proprietary x86 applications on ARM-based systems.

For security research specifically, Unicorn Engine provides a lightweight multi-architecture emulator based on QEMU but designed for fine-grained control over CPU state and memory. In contrast, Icicle represents a newer approach emphasizing performance through JIT compilation, though it currently supports a more limited instruction set.

3.2 Testing Methodologies

Current approaches to emulator validation focus predominantly on functional testing—verifying that programs execute correctly through CPU stress tests, application compatibility suites, and standard benchmarks like SPEC CPU. Performance benchmarking complements this by measuring execution speed, throughput, and latency. However, neither

approach systematically evaluates the correctness of subtle CPU behaviors that occur under unusual conditions.

Security-focused testing remains fragmented. Some research documents anti-emulation techniques used by malware, and occasional papers explore vulnerabilities in specific emulators. For example, projects like `afl-unicorn` combine AFL fuzzing with Unicorn Engine for targeted binary fuzzing, showing interest in leveraging emulation for security testing. Nevertheless, this gap means that emulator developers lack regression tests for subtle behaviors, and security practitioners lack evidence-based guidance for choosing appropriate emulation tools.

4 Problems and Opportunities

4.1 Identified Problems

Our preliminary investigation shows inconsistencies in how emulators handle edge cases. For instance, for FPU stack overflow, native hardware and Blink correctly detect the condition (returning status 0x3a41), while QEMU, Box64, and Unicorn fail to set the overflow flags (returning 0x3800). Similarly, Blink exhibits an anomalous bug in LAHF flag handling, incorrectly setting the Auxiliary Flag to yield 0x0b instead of the correct 0x03. Additionally, the RDTSCP instruction reveals another discrepancy: native hardware returns non-zero processor IDs through TSC_AUX, while Blink consistently returns zero, thereby enabling trivial emulator detection.

These discrepancies mean no authoritative "ground truth" exists beyond native hardware execution. Consequently, security products that rely on emulators for malware sandboxing operate with blind spots. For example, antivirus systems using QEMU may miss the FPU-based evasion techniques, malware analysis frameworks can be trivially fingerprinted through behavioral testing, and exploit proof-of-concepts developed in emulated environments may fail on actual hardware.

During our testing, we discovered a vulnerability in Unicorn Engine: an 80KB heap buffer overflow triggered by architecture mismatches in context restoration. Significantly, this vulnerability shows that emulators themselves are potential attack surfaces.

4.2 Research Opportunities

Creating a comprehensive test suite for CPU emulator edge cases would provide a reproducible methodology for comparing emulators and enable automated testing across versions. Evidently, such a framework would give security researchers evidence-based guidance for choosing appropriate emulators, whether for malware analysis, exploit development, or fingerprint detection.

For emulator developers, comprehensive test cases would enable regression testing and help prioritize which behaviors require fixing. Beyond this, our vulnerability discovery methodology can be applied beyond x86-64 to other architectures like ARM and RISC-V, potentially uncovering similar issues in their respective emulation frameworks. Importantly, no existing research evaluates emulator accuracy for security-relevant CPU behaviors, thereby making this work valuable for practitioners choosing tools, developers

improving implementations, and organizations deploying emulation-based security infrastructure.

5 Project Overview and Architecture

5.1 Framework Design

Our testing framework addresses the challenge of comparing emulator behaviors while accommodating their diverse interfaces and implementations. Specifically, the design uses three principles: trait-based abstraction provides a common interface while supporting emulator-specific implementations, modular separation decouples test cases from emulator runners and analysis logic, and deterministic test cases ensure reproducibility.

The core abstraction is the `EmulatorRunner` trait, defining methods for instantiating an emulator, executing shellcode with instruction limits, reading register values, and identifying the emulator. This design allows writing generic tests that work across all conforming implementations, thereby isolating emulator-specific details behind the trait boundary.

We developed three fundamental test patterns examining behaviors often misimplemented in emulation. For instance, the FPU stack overflow test pushes nine values onto an eight-register stack to verify exception handling. Meanwhile, the LAHF flag test examines whether auxiliary flag bits are set correctly after operations. In addition, the RDTSCP test reads the timestamp counter and processor ID to check if emulators properly model processor identification. Each test includes not just shellcode but documentation of expected results from native hardware and analysis of security relevance.

Emulator implementations range from direct native execution (establishing ground truth) to subprocess invocation for tools like Blink and QEMU, to FFI bindings for libraries like Unicorn and Icicle. As expected, each implementation must handle its particular quirks. For instance, Blink returns results through exit codes, QEMU lacks direct register access, Box64 requires pseudo-terminal environments, and Unicorn needs careful memory mapping. Subsequently, the analysis engine compares all emulator results against native hardware, categorizing discrepancies and documenting their security implications.

5.2 Test Case Implementation

The FPU stack overflow test executes eight `f1dz` instructions to fill the FPU register stack, then attempts a ninth push. On hardware, this triggers both the Invalid Operation and Stack Fault flags, thereby yielding a status word of `0x3a41`. The test reads this status using `fstsw ax` and returns it in RAX. This behavior matters because exploits may rely on FPU exceptions to detect buffer overflows or manipulate heap metadata through controlled exception handling.

The LAHF flag test performs a simple operation sequence: clear the AL register with `xor al, al`, execute `lahf` to load flags into AH, then zero-extend AH to RAX for return. The expected value of `0x03` reflects the reserved bit 1 being set while other flags remain clear. Importantly, deviations from this pattern enable emulator fingerprinting, as malware can detect it's running in an analyzed environment by checking for known flag discrepancies.

The RDTSCP test reads both the timestamp counter and the TSC_AUX register, which contains processor identification information. Native hardware returns non-zero values identifying the current CPU core, whereas Blink consistently returns zero for TSC_AUX. Consequently, this discrepancy allows reliable fingerprinting of emulated execution environments.

6 Implementation and Technical Challenges

6.1 Implementation Challenges

Implementing runners for diverse emulators reveals fundamental differences in how they expose execution capabilities. For instance, native hardware execution required mapping executable memory pages and casting shellcode buffers to function pointers, a straightforward approach that nonetheless must contend with W^X security policies that prevent simultaneous write and execute permissions. We addressed this using `memfd_create` and `fexecve` to create properly-permissioned executable regions while isolating potentially-crashing shellcode in subprocesses.

Blink and QEMU create a different challenge. They execute complete binaries and communicate results through exit codes. This works acceptably for small return values but truncates 64-bit results, thereby forcing us to design test cases that return meaningful data in the lower eight bits. In fact, we discovered Blink’s Auxiliary Flag bug through this process. The emulator consistently returned 0x0b instead of the expected 0x03, thus revealing an implementation error in its flag handling logic.

Box64 introduced an unexpected requirement: proper execution demands a pseudo-terminal environment. Without PTY support, Box64 either hangs indefinitely or produces incorrect results.

Unicorn Engine, despite providing the most control through its C library API, had the most serious issue. During testing of context save and restore functionality, we triggered an 80KB heap buffer overflow caused by architecture mismatches. Specifically, the context size calculation uses the source architecture (potentially ARM with large context structures), but copies into a destination buffer sized for a different architecture (x86 with smaller structures).

Each emulator implementation reveals something about the inherent complexity of accurate CPU emulation. For instance, Icicle’s limited instruction support meant some tests couldn’t execute at all. Moreover, QEMU’s TCG, used by derivative projects like Unicorn, propagates bugs across implementations. As it turns out, the FPU overflow detection failure appears in QEMU, Box64, and Unicorn, though each implements the x87 FPU independently, thereby suggesting this is a commonly overlooked edge case rather than shared code.

7 Status and Results

7.1 Experimental Results

We tested eight emulation environments across three edge case scenarios, using native x86-64 hardware (Intel and AMD processors running Linux) as ground truth. All emulators

were tested at their latest stable releases.

The FPU stack overflow test shows a clear pattern. Only Blink correctly implements overflow detection, matching native hardware’s 0x3a41 status word. In contrast, QEMU, Box64, and Unicorn all return 0x3800, indicating they fail to set the Invalid Operation and Stack Fault flags. Meanwhile, Icicle returns 0x0000, suggesting no FPU support at all. Additionally, MWEMU and KUBERA handle the overflow differently: MWEMU detects the condition but panics instead of setting flags, while KUBERA does not support FPU instructions.

Conversely, the LAHF flag test shows Blink with a unique bug while other emulators succeed. Native hardware and all emulators except Blink return 0x03, but Blink returns 0x0b. Specifically, the bit pattern difference (00001011 versus 00000011) reveals that Blink incorrectly sets the Auxiliary Flag at bit 4. While the AF flag sees limited use in modern software outside BCD arithmetic, this discrepancy nevertheless enables trivial emulator fingerprinting. In reality, malware can reliably detect Blink execution by performing a simple LAHF test and checking for the anomalous bit pattern.

RDTSCP behavior shows a clear distinction between native hardware and emulation. Native hardware returns non-zero processor IDs through the TSC_AUX register, identifying the current CPU core executing the instruction. In contrast, Blink consistently returns zero for TSC_AUX, thereby failing to implement processor identification. Consequently, this consistent zero return enables fingerprinting where malware can distinguish Blink emulation from native execution by examining processor identification behavior.

7.2 Discovered Vulnerabilities

When doing some testing on Unicorn Engine’s context management API, we discovered a heap buffer overflow. Specifically, the vulnerability occurs in the context restoration function when handling architectures with differing CPU state sizes. The size calculation uses the source architecture’s context size (potentially 80KB for ARM with its extensive register set) but copies this data into a destination buffer allocated for the target architecture, which may be only 4KB for x86. Consequently, this mismatch causes a heap overflow that reliably crashes Unicorn-based applications.

8 Planning for Next Semester

8.1 Expanding Test Coverage

The framework’s current three test cases work but barely scratch the surface of CPU behavioral complexity. Over the next semester, we plan to expand coverage to reveal more emulator discrepancies. Priority areas include memory ordering and atomics (LOCK prefix, fence instructions), segment register boundary conditions (particularly FS and GS base address handling), CPUID topology enumeration and feature bit reporting, timer precision measurements, and AVX/AVX2 vector instruction handling with optional AVX-512 support where available.

Adding test cases incrementally allows us to build a comprehensive suite while maintaining quality. Specifically, each new test requires careful analysis of native hardware behavior, implementation across all emulator runners, and documentation of security implications.

Consequently, this approach ensures that every test provides actionable information for emulator selection and improvement decisions.

8.2 Sandsifter Integration

Beyond manual test case development, we plan to integrate and optimize the Sandsifter x86 fuzzer [2]. Remarkably, Sandsifter exhaustively searches the x86 instruction space for undocumented or incorrectly implemented instructions through comprehensive fuzzing of the instruction set to discover hidden behaviors and implementation flaws. Applying this tool to our emulator suite could reveal additional behavioral discrepancies beyond those we've manually identified. In addition, the fuzzer's thorough approach complements our targeted corner case testing, potentially uncovering unexpected instruction interactions or rarely-exercised code paths in emulation engines.

Complementary approaches like afl-unicorn, which combines AFL fuzzing capabilities with Unicorn Engine, show the potential of fuzzing-based emulator testing. While afl-unicorn focuses on fuzzing binaries through emulation, adapting similar coverage-guided techniques to fuzz emulator implementations themselves could nevertheless reveal corner cases missed by manual test development.

8.3 Upstream Contributions

Working with open-source emulator maintainers is an important part of improving the broader emulation ecosystem. Specifically, for QEMU TCG, Box64, Icicle, and MWEMU, we aim to contribute patches addressing the missing x87 overflow detection where technically feasible. Similarly, Blink's AF and RDTSCP bugs require targeted fixes that we can provide with proof-of-concept patches demonstrating correct behavior.

The Unicorn Engine vulnerability requires coordinated disclosure following responsible security practices. We're working with maintainers to validate patches for the `uc_context_restore()` validation issue, thereby ensuring the fix addresses the root cause without introducing new problems. Subsequently, following the 90-day embargo, we'll publish technical details and proof-of-concept code to help other Unicorn users assess their risk and apply updates if needed.

9 Conclusion

9.1 Summary

This research evaluates CPU emulator accuracy for security-relevant corner cases. Through a trait-based testing framework in Rust, we compared eight widely-used x86-64 emulators against native hardware across three fundamental instruction patterns. Our findings show that every tested emulator exhibits behavioral discrepancies, with most failing to detect FPU stack overflow despite its security relevance for exploit development and analysis.

9.2 Implications

Security researchers should validate findings on native hardware whenever possible, particularly for boundary conditions involving exception handling or subtle CPU state. More-

over, emulator developers should incorporate these test cases as regression tests, thereby ensuring that accuracy improvements don’t regress with future changes. In addition, organizations deploying emulation-based security infrastructure should understand their chosen emulator’s limitations and consider multiple emulation strategies for defense-in-depth.

9.3 Closing Thoughts

CPU emulation pervades modern security research, from malware sandboxes to cross-architecture exploit development. As demonstrated, this work shows that all emulators have behavioral quirks that can mislead researchers or enable detection by sophisticated adversaries. By providing rigorous evaluation methodology, documented results, and an extensible testing framework, we enable the community to make informed choices and improve emulation accuracy incrementally.

Security infrastructure needs the same rigorous validation we apply to other systems. As emulation becomes increasingly central to security workflows, understanding its limitations and improving its accuracy becomes essential for reliable security research.

References

- [1] W. Chen and P. Liu. Anti-emulation techniques in malware analysis. In *IEEE Symposium on Security and Privacy*, pages 215–230. IEEE, 2018.
- [2] C. Domas. Breaking the x86 instruction set. Technical report, Cyber Security Research Lab, 2017. Sandsifter: The x86 processor fuzzer for hidden instructions and hardware bugs.
- [3] MITRE Corporation. Cve-2018-3665: Lazy fp state restore vulnerability. *Common Vulnerabilities and Exposures*, 2018.
- [4] MITRE Corporation. Cve-2021-44078: Unicorn engine sandbox escape via split_region memory comparison flaw. <https://nvd.nist.gov/vuln/detail/CVE-2021-44078>, 2021. CVSS 8.1 (HIGH). Flaw in virtual memory manager’s split_region function using incorrect block->offset comparison. Fixed in version 2.0.0-rc5. Accessed: 2026-02-02.
- [5] T. Ormandy. Analysis and exploitation of an eset vulnerability. <https://googleprojectzero.blogspot.com/2015/06/analysis-and-exploitation-of-eset.html>, 2015. Google Project Zero. Remote root exploit via ESET’s x86 emulator used for malware scanning.