Instruction-Level Exploratory Testing Framework for ARMv8-A Processors

Yang Xinzhe, Chai Song, Chen Xi, Gong Min

Southwest Minzu University, Chengdu, Sichuan, 610041, China

Abstract: Undocumented behavior in modern processors is difficult to evaluate because the encoding space is huge and the expected outcome is often only partially specified. This study focuses on ARMv8-A and builds an instruction-level exploratory testing framework for architecturally undefined encodings. We first prune the 32-bit opcode space with an ARM-specification-aware decoder, then execute the remaining candidates on an RK3399 platform through an isolated multi-process harness. Rather than forcing every case to match a golden model, we characterize each executable candidate with three weak oracles: register-state differencing, PMU observations, and sentinel-guarded control-flow checks driven by signal contexts. The full campaign covers 1,203,581,952 undefined encodings and identifies 105,738,315 executable candidates. Within that executable set, non-trivial subsets produce silent register corruption or implicit memory activity, whereas no silent program-counter deviation is observed for the tested undefined branches. These results suggest that undocumented instruction behavior on commercial ARMv8-A silicon is not random noise; it can be screened exhaustively and organized into stable behavioral classes even when complete reference semantics are unavailable.

Keywords: ARMv8-A; Undocumented instruction; Instruction-level testing; Weak oracle; Control-flow verification

DOI: 10.62639/sspjinss09.20260301

1. Introduction

ARMv8-A processors expose a large 32-bit encoding space, but only part of that space is architecturally defined by the official manual and associated machine-readable specifications[1]. Everything outside the documented region is usually treated as "should not execute", yet the hardware decoder still has to process those bit patterns. For post-silicon validation, that gap matters. Once an implementation handles an undefined encoding in a stable but undocumented way, the real question is no longer whether the instruction can run, but what state it leaves behind and whether it disturbs control flow or memory state.

Existing exploration tools largely treat the problem as a discovery task: find which hidden or undocumented encodings execute at all. Sandsifter on x86[3] showed that exhaustive instruction mining is feasible, and later RISC-oriented scanners such as IScanU[4], Armshaker[5], UISFuzz[8], and Ins Finder[7] improved the coverage or speed of the search. Those tools are valuable for mapping the space, but their main output is still a hit list. They say much less about the architectural side effects that appear after execution, and that is exactly where validation and reliability concerns begin.

A second line of work compares hardware against emulators or other reference implementations. Examiner, for example, locates inconsistent ARM instructions between real devices and CPU emulators[6]. That approach is effective when the reference side is trustworthy. Undefined encodings, however, sit precisely where a complete model is least

(Manuscript NO.: JINSS-26-1-62013)

About the Author

Yang Xinzhe (2001-), male, Han ethnicity, from Chengdu, Sichuan, China, currently a master's student, His research direction is embedded hardware security.

Funding

This work was supported by the Sichuan Provincial Science and Technology Support (Program under Grant 2024ZHCG0194).

reliable. In that sense, the problem is closer to the broader oracle problem in software testing: expected results are partial, weak, or missing[2]. We therefore study undocumented instructions as a weak-oracle testing problem and rely on direct observations collected from real hardware rather than on a full semantic model.

Based on that view, we build an instruction-level exploratory testing framework for ARMv8-A. The framework filters out defined instructions with a specification-aware decoder, executes the remaining candidates on an RK3399 platform in isolation, and classifies each executable case through three observations: register-state differencing, PMU events, and control-flow checks driven by signal contexts. The goal is not to recover the exact semantics of every undocumented instruction. Instead, we aim to separate stable behavior classes from noise and to identify cases that deserve closer reliability analysis.

2. Related Work

(1) Instruction-space scanning

Undocumented-instruction studies often begin with enumeration. Domas used Sandsifter to traverse the x86 instruction space and demonstrated that hidden regions can be found automatically at scale[3]. On ARM and other RISC platforms, later tools pursued the same objective with architecture-specific pruning or faster scanning strategies, including IScanU[4], Armshaker[5], UISFuzz[8], and Ins Finder[7]. Their common strength is broad coverage. Their common limitation is that an instruction is usually counted as interesting as soon as it executes, even if the resulting architectural state is still unclear.

(2) Differential testing and weak oracles

Differential frameworks try to go further by comparing hardware with software models. Examiner targets inconsistent ARM instructions between silicon and CPU emulators[6], which is useful when the emulator is mature enough to serve as a credible baseline. For undefined encodings, that assumption becomes fragile because the missing specification often propagates directly into the model. The absence of a fully reliable oracle is therefore not a side issue; it is the center of the problem.

Barr et al. describe the oracle problem as the difficulty of deciding whether program behavior is correct when expected outputs are unavailable, incomplete, or too expensive to obtain[2]. Our work adopts that perspective. Instead of demanding a full semantic judgment for every undocumented opcode, we ask a narrower but operational question: does the instruction leave reproducible evidence in registers, PMU counters, or control-flow state? That framing turns undocumented instruction analysis into a practical exploratory testing task.

3. Methodology

The framework is organized as a three-stage pipeline. We first construct the candidate corpus, then perform automated large-scale screening, and finally apply multi-dimensional characterization to the opcodes that survive the screening stage. The same pipeline is used throughout the campaign so that results remain comparable across repeated runs.

(1) Candidate corpus generation

Each test campaign starts from the full 32-bit AArch64 encoding space. Architecturally defined instructions are removed with a disassembler built from ARM machine-readable specifications and checked against the ARM architecture manual[1]. This static filtering step reduces 4.29 billion possible encodings to 1,203,581,952 architecturally undefined candidates. The decoder is used only to determine what is outside the published specification; after that point, all behavioral evidence comes from direct hardware execution.

(2) Automated screening harness

The screening stage is organized as a dispatcher-worker pipeline. The dispatcher enumerates candidate opcodes and assigns them to worker processes pinned to the Cortex-A53 cluster on the RK3399. Each worker copies one opcode into an executable page, wraps it in a minimal harness, and runs it inside a process boundary that can be restarted after faults. This design keeps the global campaign moving even when individual candidates terminate through SIGILL, SIGSEGV, or other exceptions.

Two implementation choices are important for measurement fidelity. First, the entry and exit code around the test snippet is kept intentionally small so that the framework does not overwrite the same registers it wants to observe. Second, parent and child exchange data through shared memory rather than text-oriented IPC. That decision reduces overhead and makes large-scale screening practical without sacrificing context information.

(3) Multi-dimensional functional characterization

Execution alone is not treated as a sufficient result. After the screening stage admits an opcode, we record three classes of evidence. The register oracle compares pre- and post-execution contexts to catch silent corruption in general-purpose registers. The PMU oracle checks whether the instruction triggers counter changes consistent with hidden memory activity or other non-trivial work. The control-flow oracle verifies that the program counter lands where it should by combining a sentinel-guarded layout with trap-based fall-through detection.

These observations are deliberately weak: none of them claims to reconstruct the full semantics of an undocumented instruction. Together, however, they are strong enough to separate benign no-ops from candidates that modify architectural state, touch memory implicitly, or try to leave the expected control path. This is the level of evidence we need for exploratory testing on real hardware.

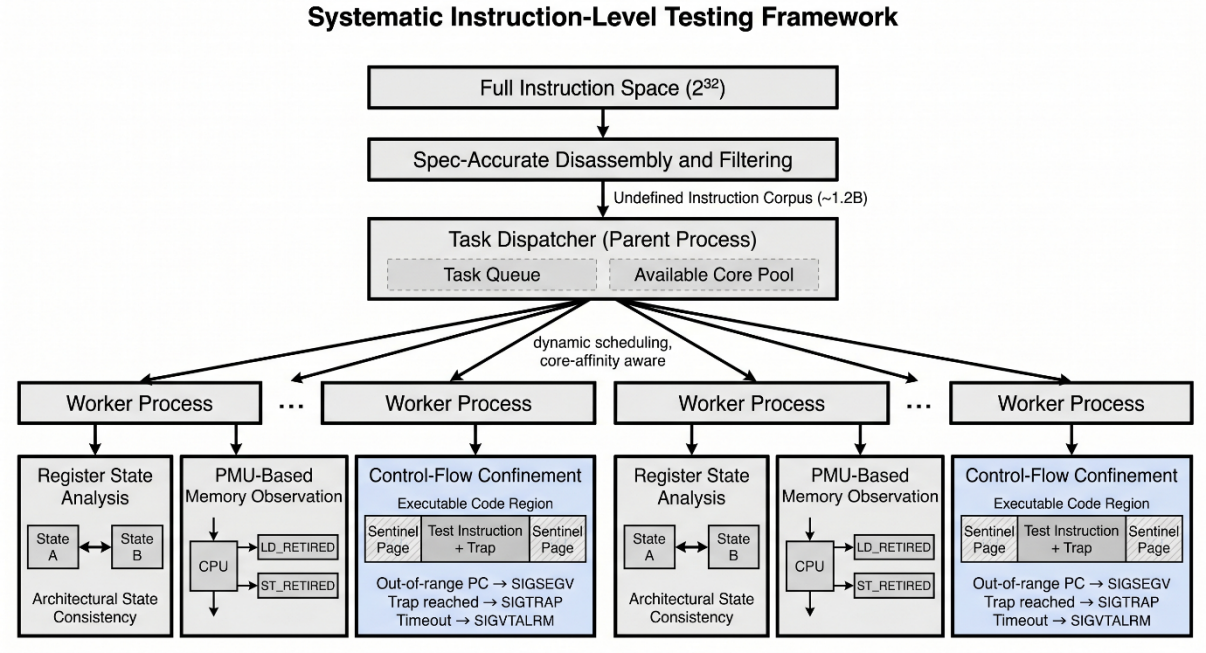

Figure 1. Workflow of the proposed instruction-level exploratory testing framework.

4. Implementation

(1) Platform configuration and isolation

Experiments were performed on an RK3399 SoC containing four Cortex-A53 cores and two Cortex-A72 cores. To

reduce cross-core noise, the campaign was bound to the Cortex-A53 cluster and executed under a fixed software environment. Process isolation, explicit CPU affinity, and controlled restart behavior were used to keep exception handling from leaking into later tests.

(2) Harness realization

The worker harness is implemented as a multi-process system. A parent process dispatches candidate encodings, while child processes execute them in dedicated pages with execute permission. A compact assembly stub captures the context before and after execution, and shared-memory buffers return register snapshots and auxiliary counters to the parent without repeated serialization.

Because undocumented instructions can terminate in several ways, the harness treats signals as data rather than as fatal errors. Exceptions are intercepted, classified, and folded back into the result record for the current candidate. This makes it possible to distinguish a genuinely executable opcode from one that merely reaches an exception immediately.

(3) Oracle realization

The register oracle saves the general-purpose context on a dedicated stack region and compares it after the test instruction returns or traps. The PMU oracle reads selected performance counters around the same execution window to capture hidden activity that may not appear in the visible register difference. For control flow, the harness places sentinels around the expected fall-through path and uses signal contexts to measure program-counter deviation. In our experiments, this design was sufficient to tell apart ordinary fall-through, explicit traps, and unexpected control-flow outcomes.

5. Experimental Results

(1) Screening efficiency and coverage

The exhaustive campaign covered all 1,203,581,952 candidates in the undefined 32-bit encoding space. Using four Cortex-A53 workers, the full scan finished in about 7.4 hours. Average throughput was approximately 11,300 instructions per second per core, or about 45,200 instructions per second for the configured system.

We repeated the screening phase four times to check stability. Three runs agreed within five opcodes, which indicates that the harness was deterministic enough for large-scale enumeration. For subsequent analysis, we keep the maximum stable yield: 105,738,315 executable undocumented opcodes, corresponding to 8.79% of the candidate set.

(2) Observed behavior categories

Existence alone is not the most useful result. Within the executable undocumented set, 12.77% of the candidates caused detectable silent corruption in general-purpose registers, and 3.01% produced PMU evidence consistent with implicit memory traffic. These cases deserve more attention than simple no-ops because they can perturb software state without first surfacing as an illegal-instruction exception.

The control-flow checks show a different picture. Across the tested undefined branches, we did not observe silent program-counter deviation: control either returned along the expected path or terminated through an explicit exception path that the harness could account for. On the evaluated Cortex-A53 implementation, exception isolation for undefined control transfers therefore appears stronger than the corresponding data-path behavior.

(3) Limitations and validity

The present study is limited to one commercial platform and to the specific weak oracles implemented in the

harness. Some undocumented behaviors may depend on privilege state, surrounding instruction context, micro-architectural state, or platform configuration. We therefore interpret the current results as a stable characterization of RK3399/Cortex-A53 behavior rather than as a universal statement about all ARMv8-A implementations.

6. Conclusion

This study turns the analysis of undocumented ARMv8-A instructions into a large-scale exploratory testing problem. By screening the entire undefined 32-bit encoding space on RK3399 and then classifying executable cases with weak oracles, we show that undocumented behavior is both widespread and structured rather than purely accidental.

The most practical takeaway is that invisible data-path effects are more common than silent control-flow escapes on the evaluated platform. Future work should extend the same harness to additional cores, privilege states, and surrounding contexts so that the undocumented behaviors reported here can be compared across implementations and eventually fed back into validation rules and architectural documentation.

References

[1] ARM LTD. Arm Architecture Reference Manual for A-profile architecture[M/OL]. Cambridge: Arm Ltd., 2022[2026-04-13]. Available: https://developer.arm.com/documentation/ddi0487/latest.

[2] BARR E T, HARMAN M, MCMINN P, et al. The oracle problem in software testing: A survey[J]. IEEE Transactions on Software Engineering, 2015, 41(5): 507-525.

[3] DOMAS C. Breaking the x86 ISA[C/OL]//Black Hat USA 2017. Las Vegas, NV, USA, 2017[2026-04-13]. Available: https://www.blackhat.com/us-17/briefings.html#breaking-the-x86-isa.

[4] DOFFERHOFF R, GOEBEL M, RIETVELD K, et al. IScanU: A portable scanner for undocumented instructions on RISC processors[C]//2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Valencia: IEEE, 2020: 306-317.

[5] STRUPE F, KUMAR R. Uncovering hidden instructions in Armv8-A implementations[C]//Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy. New York: ACM, 2020: 1-9.

[6] JIANG M, XU T, ZHOU Y, et al. EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM[C]//Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2022: 846-858.

[7] DONG R, CUI B, SUN Y, et al. Ins Finder: A practical CPU undocumented instruction detection framework[J]. Journal of Circuits, Systems and Computers, 2024, 33(3): 2450047.

[8] LI X, WU Z, WEI Q, et al. UISFuzz: An efficient fuzzing method for CPU undocumented instruction searching[J]. IEEE Access, 2019, 7: 149224-149236.