

The Residual Nondeterminism Gap in Forensic Record-and-Replay

Jason L. Wright (✉)^[0000–0002–6443–1681] and Thomas L. Baldwin^[0009–0007–9273–6291]

Idaho State University, Pocatello, ID 83209, USA
wrigjaso@isu.edu, baldthom@isu.edu

Abstract. Record-and-replay is foundational to digital forensic analysis, enabling investigators to faithfully re-execute recorded program behavior for post-incident examination. These systems assume the CPU is deterministic: given identical inputs, an instruction produces identical results. Modern x86 processors violate this assumption through hardware entropy instructions (`RDRAND` and `RDSEED`; analogous instructions exist on other ISAs) that return unpredictable values at any privilege level with no existing mechanism for per-process interception. While other sources of instruction-level nondeterminism have been addressed by hardware mitigations (UMIP, CPUID faulting, `CR4.TSD`), entropy instructions remain a blind spot. We present a taxonomy that distinguishes environmental from entropic nondeterminism, document three concrete failure mechanisms in the widely deployed `rr` record-and-replay system, and show that the scope of the gap has expanded as libraries silently extend their `RDRAND` dispatch to new hardware platforms. We analyze the implications for forensic evidence integrity (including silent trace divergence when entropy-derived values affect only data, and adversarial exploitation) and survey the design space for solutions. The gap we characterize affects any process-level record-and-replay system that captures nondeterminism at the syscall boundary, and represents a concrete threat to the reliability of replay-based forensic analysis.

Keywords: Record-and-replay · Digital forensics · Deterministic execution · Hardware entropy · `RDRAND` · Nondeterminism.

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-032-35579-9_33. Use of this Accepted Version is subject to the publisher’s Accepted Manuscript terms of use <https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms>.

First published in *Availability, Reliability and Security. ARES 2026 International Workshops*, LNCS vol. 16901, pp. 614–632, 2026, by Springer Nature.

1 Introduction

Digital forensic analysis increasingly depends on the ability to faithfully re-execute recorded program behavior. After a sophisticated attack or data breach, investigators must reconstruct the sequence of events that led to the compromise, often weeks or months after the fact. Record-and-replay systems address this need by capturing the nondeterministic inputs to a program during execution and substituting them during re-execution, enabling the investigator to observe identical behavior across multiple replay passes [9]. This capability is not merely a convenience: it is the technical foundation upon which forensic replay tools such as PANDA [16] and ReVirt [17] are built.

The correctness of replay-based forensic analysis rests on a critical assumption: that the CPU is deterministic. Given identical register and memory state, an instruction must produce identical results. If this assumption is violated, the replayed execution will diverge from the original, and the investigator’s observations will not reflect what actually happened.

Modern processor architectures expose a class of instructions that violate this assumption by design. The RDRAND and RDSEED instructions on x86 [24, 1], and analogous instructions on other ISAs (Section 2.2), return hardware-generated entropy at any privilege level with results designed to be non-reproducible. A binary that executes such an instruction during a forensic recording may produce a trace that cannot be faithfully replayed without an out-of-band interception mechanism.

The systems community has not ignored instruction-level nondeterminism. Hardware vendors and OS developers have introduced mechanisms to trap several categories of nondeterministic instructions: User-Mode Instruction Prevention (UMIP) restricts descriptor table queries, CPUID faulting [31, 35] intercepts feature discovery, and CR4.TSD [24] traps timestamp counter reads. Together these mechanisms address a substantial fraction of the nondeterministic instruction landscape. Yet no existing mechanism provides per-process interception of hardware entropy instructions.

Individual symptoms of this gap have been noted in the `rr` project’s issue tracker, wiki, and source comments; this paper makes four contributions that go beyond those scattered observations to characterize the gap as a structural property of syscall-boundary replay, measure its scope across the deployed software ecosystem, analyze its forensic implications, and identify a concrete architectural path to closing it.

First, we develop a taxonomy distinguishing *environmental* from *entropic* nondeterminism and show that existing hardware mitigations address the former but leave the latter unhandled (Section 3). The framing applies to any syscall-boundary replay system: the gap is structural, not a defect of any single tool.

Second, we systematize three failure mechanisms in the widely deployed `rr` [32] system: library-internal RDRAND usage, compile-time emission via compiler intrinsics, and the AMD CPUID-faulting gap that left `rr` inoperative on AMD hardware for nearly a decade (Section 4). The first and third have been

discussed individually in the `rr` issue tracker; we add the second, whose latent exposure has not previously been documented.

Third, we quantify the gap’s scope across 33,909 ELF binaries from six Debian releases (2015–2025) and a Gentoo installation, of which 99 contain `RDRAND` or `RDSEED`, classified by `CPUID` gating via disassembly (Section 4.3). All conform to a caller-side `CPUID` convention that is unverifiable without per-binary reverse engineering and trivially bypassable through documented compiler flags.

Fourth, we analyze the forensic implications (silent divergence, adversarial exploitation, and latent compile-time emission risk) and survey the design space, identifying hypervisor-based per-process interception built on existing KVM infrastructure as the most promising near-term path on Intel hardware (Sections 5 and 6).

This paper is a position paper: our contribution is problem characterization and empirical grounding, not a complete system implementation. The design, implementation, and evaluation of a solution are the subject of ongoing work.

2 Background

2.1 Record-and-Replay for Digital Forensics

Record-and-replay is a technique for achieving deterministic re-execution of programs that interact with nondeterministic environments. The core insight is that a sequential computation on a deterministic processor is itself deterministic: only the inputs crossing the process boundary (system calls, signals, device I/O) need to be recorded [9]. During replay, logged inputs are substituted for live inputs, and the processor reproduces the original execution faithfully.

VM-level systems (ReVirt [17], Aftersight [10], PANDA [16] on QEMU [6]) log all guest inputs and achieve completeness at the cost of significant overhead; Küchler et al. [26] demonstrate the cost at scale by analyzing 100,000 malware samples under PANDA. Process-level systems narrow the logging boundary to a single process, dramatically reducing overhead; Mozilla’s `rr` [32] is the most widely deployed example, and Quinn and Alvaro [38] provide a recent overview emphasizing precise identification of nondeterministic actions over wholesale logging. Section 7 treats both families in more detail.

VM-level emulation can intercept `RDRAND` during instruction translation, at the cost of whole-VM overhead; process-level systems rely on native execution and are structurally blind to instructions that complete unconditionally at any privilege level, raise no fault, and generate no syscall. We focus on the process-level regime (Section 4, Table 5).

2.2 Hardware Entropy Instructions

The Intel 64 and IA-32 Architectures Software Developer’s Manual [24] and the AMD64 Architecture Programmer’s Manual [1] define `RDRAND` and `RDSEED` as instructions that return hardware-generated entropy directly to a general-purpose

register. `RDRAND` draws from a deterministic random bit generator (DRBG) seeded by a hardware entropy source, while `RDSEED` provides direct access to conditioned entropy from the underlying hardware noise source. Both are available at all privilege levels.

Unlike `RDTSC` (which reads a counter that could in principle be controlled) or `CPUID` (which returns stable values that could be emulated), hardware entropy instructions return values that are by architectural definition unpredictable and non-reproducible. No environment configuration can make `RDRAND` return a predictable value, because the instruction is specifically designed to defeat such predictions.

The architectural pattern is not unique to x86. ARM v8.4+ introduces the `FEAT_RNG` extension, exposing hardware entropy through the system registers `RNDR` and `RNDRRS` [4]; reads complete at any exception level absent `FEAT_RNG_TRAP` and return values that are, like `RDRAND`, non-reproducible by design. We focus on x86 in this work because the deployed process-level record-and-replay systems (notably `rr`) target x86, but the gap generalizes to any ISA exposing hardware entropy via unprivileged, non-trapping instructions.¹

2.3 Threat Model and Scope

We consider a forensic analyst who records a target binary’s execution under a process-level record-and-replay system (`rr` being the only widely deployed example) and replays the trace to reconstruct program behavior. The analyst controls the recording environment (kernel, replay tool, CPU vendor) but not the target binary; faithful replay requires that register and memory state during replay match the recorded execution.

Target binaries fall into four classes (Table 1). *Dynamically linked binaries* that follow the `CPUID`-then-`RDRAND` convention are the best-supported case: `rr`’s instruction patching covers library `RDRAND`/`RDSEED` sites, and on Intel, `CPUID` suppression additionally causes well-behaved libraries to skip the path entirely. *Statically linked binaries* embed `RDRAND` in the main text segment outside the instruction-patching window; on Intel, `CPUID` suppression still gates the path, but on AMD prior to Linux 6.17 no mechanism applies and replay either diverges silently or aborts at the next syscall boundary (Section 4). *Stripped or adversarial binaries*—the analyst’s primary forensic target—may bypass the `CPUID` convention entirely, either through compile-time `RDRAND` emission (Section 4.3) or by deliberate adversarial choice; no existing process-level mechanism reliably covers this class. *VM-level workloads* are out of scope: VM-granularity replay systems can intercept `RDRAND` during instruction translation, accepting whole-VM overhead as the cost of completeness; our focus is the low-overhead process-level regime where this option is unavailable.

¹ Hardware bugs compound the problem: AMD SB-7055 (CVE-2025-62626) reports that 16-bit and 32-bit `RDSEED` on Zen 5 may return zero while incorrectly reporting success [2], introducing divergence even when the replay tool intercepts the instruction.

Table 1. Target-binary classes and the `rr` 5.9.0 mechanisms that handle their `RDRAND`/`RDSEED` usage. `CPUID` suppression requires hardware `CPUID` faulting, available on Intel since Linux 4.12 (2017) and on AMD since Linux 6.17 (2025). Instruction patching intercepts `RDRAND`/`RDSEED` bytes in dynamically loaded shared libraries only.

Target class	Primary mechanism	Coverage
Benign, dynamic linkage	Patching + <code>CPUID</code> supp.	Strong
Benign, static linkage	<code>CPUID</code> suppression only	Intel only
Stripped/static, adversarial	<code>CPUID</code> supp. (bypassable)	None
VM-level workload	Out of scope (VM-granularity replay)	

Table 2. Nondeterministic x86 instructions and their existing per-process mitigations. Instructions marked **none** constitute the residual gap. `DRBG`: deterministic random bit generator.

Instruction	Category	Source	Mitigation
<code>RDTSC</code> / <code>RDTSCP</code>	Environmental	Timestamp	<code>CR4.TSD</code>
<code>CPUID</code>	Environmental	Features	<code>CPUID</code> faulting
<code>SGDT</code> et al.	Environmental	Descriptors	<code>UMIP</code>
<code>RDRAND</code>	Entropic	<code>DRBG</code>	None
<code>RDSEED</code>	Entropic	Entropy src.	None

3 A Taxonomy of Instruction-Level Nondeterminism

We distinguish two categories of instruction-level nondeterminism based on the nature of the values returned and the remedies available. This distinction is important because the two categories call for fundamentally different mitigation strategies.

Environmental nondeterminism arises from instructions that read values maintained by the OS or hardware on the process’s behalf but that are not part of its architectural state in the record-and-replay sense. `RDTSC`/`RDTSCP`, `CPUID`, and the descriptor-table reads (`SGDT`, `SIDT`, `SLDT`, `SMSW`, `STR`) [24, 1] all fall here (Table 2): they can be handled by substituting a controlled environment (fixed processor model, deterministic clock, controlled `CPUID` leaves), and the existing mitigations largely accomplish this.

Entropic nondeterminism arises from instructions whose results are by architectural definition unpredictable and non-reproducible. `RDRAND` and `RDSEED` on x86 [24, 1] fall into this category: seeded by physical hardware noise, their results cannot be made predictable from prior execution state. Entropic nondeterminism requires active interception and value substitution; no environment configuration resolves it.

Table 2 summarizes the nondeterministic instruction landscape and the existing per-process mitigations for each class. Every category of environmental nondeterminism on x86 has a hardware mitigation mechanism. No category of entropic nondeterminism has one.

4 The Residual Gap: Three Failure Mechanisms

The gap identified in Table 2 is not merely theoretical. Documented failures in the `rr` project demonstrate that `RDRAND` causes replay divergence in ordinary software through three distinct mechanisms.

4.1 Library-Internal `RDRAND` Usage

System libraries invoke `RDRAND` internally without exposing any mechanism for callers to suppress the behavior. `systemd`’s `libnss_systemd` invokes `RDRAND` internally with no runtime opt-out, causing non-replayable traces on AMD systems where `rr` cannot intercept the instruction. GCC’s `libstdc++` uses `RDRAND` directly in its `std::random_device` implementation [22], and `rr` has experienced repeated breakage from this path [23].

Both libraries have since individually addressed their `RDRAND` usage, but the timelines illustrate the cost of the per-library approach. `systemd`’s internal `rdrand()` function appeared in version 240 (Debian 10, 2019); `systemd` 247 added a `SYSTEMD_RDRAND` environment variable as a partial workaround explicitly motivated by replay-debugger use, and `systemd` 251 (Debian 12, 2022) removed the function entirely, replacing all calls with `getrandom()` [44]. This three-step remediation spanned four years and two major Debian releases for a single library.

`libstdc++`’s `std::random_device` gates its `RDRAND` path behind a `CPUID` check that `rr` can suppress when `CPUID` faulting is available [22]. The recording replays faithfully, but the recorded trace does not reflect the entropy behavior the program would exhibit outside `rr`. Moreover, the exposure has been expanding: `libstdc++` 6.0.20–6.0.25 (Debian 8–10) checked only for Intel vendor strings in the dispatch, falling back to `/dev/urandom` on AMD hardware, while 6.0.28 (Debian 11) added AMD vendor detection and extended `RDRAND` dispatch to AMD for the first time [46]. The scope of `RDRAND` exposure thus grew silently with a routine library update.

`rr`’s documented approach is cooperative: it suppresses the `RDRAND` `CPUID` feature flag so that well-behaved programs believe the instruction is unavailable [32]. This defense requires that the binary consult `CPUID` before invoking `RDRAND`, a requirement the next two failure mechanisms violate.

4.2 Compile-Time `RDRAND` Emission

When C or C++ is compiled with `-mrdrnd` or `-march=native`, or Rust with `-C target-feature=+rdrand`, compiler intrinsics such as `_rdrand64_step()` emit `RDRAND` directly with no runtime `CPUID` check [21, 45]. Gentoo Linux compiles all packages from source and recommends `-march=native` as the default, implicitly enabling `-mrdrnd` on any `RDRAND`-capable host; the Rust `getrandom` crate uses the corresponding flag on Intel SGX targets (where `CPUID` faults), but nothing prevents its use on a regular Linux target. We confirmed the GCC path experimentally: `gcc -march=native -dM -E` defines `__RDRND__` on our test system,

Table 3. Prevalence of RDRAND/RDSEED instructions in ELF binaries on a Debian 13 installation (4,447 binaries scanned). All 20 binaries containing these instructions use caller-level or cached CPUID checks; none are truly ungated.

Category	Count	Gating mechanism
Crypto libraries	4	Cached CPUID init
<i>OpenSSL</i>		<i>OPENSSL_ia32_cpuid setup</i>
<i>libgcrypt</i>		<i>Co-located CPUID</i>
<i>libsodium</i>		<i>Cached global flag</i>
<i>libbotan</i>		<i>CPUID::state() singleton</i>
GCC toolchain	13	libstdc++ <i>_M_init</i> dispatch
libstdc++ (shared)	1	<i>_M_init</i> dispatch
System tools	2	Co-located CPUID
Total	20	

and the resulting binary contains RDRAND with no preceding CPUID check, identical to the explicit `-mrdrnd` build.

For all such binaries, `rr`’s CPUID suppression is irrelevant: the binary never executes CPUID to discover the feature. A test program invoking the intrinsic and compiled with `-mrdrnd` completes replay under `rr` without any error, but the RDRAND values differ silently (no signal, no diagnostic); a control build using a runtime CPUID check replays faithfully because the suppressed flag causes the check to fall through to an alternative path. Our distribution scans (Section 4.3) found no application code gaining RDRAND through compiler intrinsics in practice, but the failure mode is latent in any binary compiled with these flags.

4.3 Prevalence in Distribution Binaries

To quantify the scope of RDRAND exposure in practice, we scanned 4,447 ELF binaries across `/usr/bin`, `/usr/sbin`, `/usr/lib`, `/usr/libexec`, and `/usr/share` on a Debian 13 (Trixie) installation reproduced by the artifact’s `prevalence/debian13/Dockerfile`. Twenty binaries (0.45%) contain RDRAND or RDSEED instructions. While this percentage is small, the affected binaries are disproportionately forensically relevant: all four cryptographic libraries on the system (`OpenSSL/libcrypto`, `libgcrypt`, `libsodium`, and `libbotan`) contain RDRAND. Any binary performing TLS, encryption, or signature verification links one of these libraries. Table 3 summarizes the results.

Of the 20 binaries, 12 contain CPUID in the same function as RDRAND (directly gated); the other 8 use a cached-CPUID pattern in which a one-time initializer (`pthread_once`, `__cxa_guard`, or a static initializer) executes CPUID and stores the result in a global flag that callers test before reaching the RDRAND wrapper.

The result is that `rr`’s CPUID suppression currently handles all 20 binaries correctly: the cached flag is never set under replay, so the RDRAND path is never reached. However, this safety depends entirely on a *convention* (that RDRAND is

reachable only through caller-level checks) rather than on an architectural guarantee. We verified this convention holds across six Debian releases (8 through 13, spanning 2015–2025) and a Gentoo installation by disassembling every RDRAND-containing binary; of 33,909 ELF’s scanned across the seven distributions, 99 contain RDRAND or RDSEED, and all use caller-level or cached CPUID checks, with no exceptions. The convention’s track record is strong. The problem is that it is not *verifiable*: no toolchain enforces it, no runtime check detects a violation, and an analyst cannot confirm that an arbitrary binary respects the convention without disassembling it. For forensic purposes, a guarantee that requires per-binary reverse engineering to verify does not constitute an auditable basis for evidence integrity. The compile-time flags discussed above (`-mrdrnd`, `target-feature=+rdrand`) demonstrate that the convention *can* be bypassed trivially, even if current distribution binaries do not do so.

A scan of a Gentoo Linux system, built from source with `-march=broadwell` (gnome-light snapshot, 2026-06-02), confirms these patterns: of 3,928 ELF binaries, 13 contain RDRAND or RDSEED, and all use caller-level CPUID checks (co-located or cached). Doubling the binary count by adding the gnome-light desktop over a CLI-only Gentoo install produced zero additional RDRAND call sites: exposure is concentrated in crypto and toolchain libraries rather than spread across the user-facing package set. Despite `-march=broadwell` implicitly enabling both `-mrdrnd` and `-mrdseed` for every package, no application code gained RDRAND/RDSEED through intrinsic usage; all instances originate from library code. The current exposure is driven by a small number of core libraries rather than by widespread intrinsic use in applications, though the latent risk from compile-time emission remains.

4.4 Platform Gaps in CPUID Faulting

CPUID suppression requires hardware CPUID faulting, which on AMD only arrived in Linux 6.17 (September 2025) [35, 36]; the preceding eight-year window left `rr`’s CPUID suppression inoperative on every AMD host [42]. `rr` 5.9.0 implements a complementary instruction-patching mechanism for RDRAND/RDSEED in dynamically loaded libraries; we characterize its coverage below.

`rr`’s documentation acknowledges the gap and prescribes a per-library enumeration of workaround environment variables (`OPENSSL_ia32cap`, `SYSTEMD_RDRAND`), advising users to “find some way to avoid” RDRAND otherwise [43]. An analyst replaying AMD recordings during this period had no way to know whether a given trace was affected, because the divergence is silent.

We confirmed this gap experimentally on an AMD Ryzen 9 7940HS (Zen 4) running Linux 6.12, on which `arch_prctl(ARCH_GET_CPUID)` returns `ENOSYS`, confirming that CPUID faulting is unavailable. We recorded and replayed a program using the standard C++ API `std::random_device{}` to generate entropy, compiled without any entropy-specific compiler flags.

The dynamically linked build replayed faithfully: `rr` 5.9.0 patches RDRAND and RDSEED sites inside `libstdc++.so`. The statically linked build of the same program diverged, because the patching mechanism does not cover the main

Table 4. Replay fidelity on AMD Ryzen 9 7940HS (Linux 6.12, no CPUID faulting) under `rr` 5.9.0. Dynamic linking exposes `RDRAND` through `libstdc++.so`, which `rr` intercepts via its instruction-patching mechanism. Static linking embeds the same code in the main binary, which that mechanism does not cover, causing detectable control-flow divergence. OpenSSL 3.x provides incidental robustness through its DRBG architecture.

Program	Linkage	Entropy path	Faithful?
<code>std::random_device{}</code>	dynamic	<code>libstdc++.so</code> (intercepted)	Yes
<code>std::random_device{}</code>	static	main binary (unintercepted)	No (crash)
<code>RAND_bytes()</code>	dynamic	<code>getrandom(2)</code> via DRBG	Yes

executable’s text segment; `libstdc++`’s internal `RDSEED` retry loop (up to 99 iterations) completed a different number of times across recording and replay, and `rr` detected the tick-count mismatch at the first `write()` boundary and aborted. OpenSSL 3.x’s `RAND_bytes()` replayed faithfully through a different mechanism entirely: it seeds its DRBG via `getrandom(2)`, which `rr` records at the syscall boundary, providing *incidental* robustness that does not extend to direct `RDRAND` in statically linked code.

Table 4 summarizes the AMD experiments.

4.5 Reproducibility

All experiments and prevalence scans in this paper are reproduced by the artifact at <https://github.com/wrigjl/rdrand-replay-gap> (CC-BY-4.0, tag `v1.0`, archived at <https://doi.org/10.5281/zenodo.20599973>). The experiments execute inside a hermetic Docker image based on Debian 13 (trixie), which provides the GCC 14 and `rustc` toolchains used throughout the paper and builds `rr` 5.9.0 from source so that the version matches the one named in the text. We ran the harness on three hosts spanning Intel Haswell, Intel Raptor Lake, and AMD Zen 4 (Linux 6.12). Each experiment commits its recorded transcript and a one-line outcome summary, so that reviewers can verify the predicted outcome without running `rr` themselves. The prevalence scans (Section 4.3) are reproduced by per-distribution Dockerfiles that install the unmodified distribution package set and apply an `objdump`-based filter. Build invocations and host requirements are documented in the artifact `README`.

5 Implications for Digital Forensics

The residual nondeterminism gap has several concrete implications for digital forensic practice.

5.1 Silent Trace Divergence

Case Study: Encrypted Exfiltration Log. Consider an incident-response analyst replaying a suspected exfiltration tool under `rr` to recover the per-session XOR key it derives from `RDRAND`, intending to decrypt the matching ciphertext captured from the network and confirm what was exfiltrated.

We implemented this scenario as the program `exp3-silent` in the accompanying artifact: a 130-line C program that generates a 256-bit `RDRAND`-derived key and XOR-encrypts a fixed plaintext containing fields such as `user=admin` and `action=export_database`, emitting the ciphertext to standard output and the key to standard error. The program is deliberately written so that the `RDRAND` return values affect only data: the carry flag is ignored, the encryption loop is fixed-length, and output is emitted via fixed-width `write(2)` calls, so that the retired instruction count is identical regardless of the values returned. Compiled with `-mrdrnd` to bypass any `CPUID` check and recorded under `rr` 5.9.0 on both an Intel Haswell Xeon and an AMD Ryzen 9 Zen 4 host, the replay exits with status 0 and `rr` reports no errors on either system; the key and ciphertext, however, differ completely from those produced during recording. A practitioner reviewing only the replay output might plausibly conclude that the captured ciphertext was corrupt or that the sample is a different malware family—each an incorrect conclusion induced by the gap.

The vulnerable surface is narrower than a binary’s full text suggests: `rr` 5.9.0’s instruction-patching covers `RDRAND`/`RDSEED` in dynamically loaded libraries (Section 4), leaving only statically linked text, intrinsic-emitted application code (Section 4.3), and JIT-generated code in the silent regime. The analyst cannot determine from inspection of the recording alone whether a given binary occupied that regime.

Detectable vs. Undetectable Divergence. Whether divergence is silent depends on how `RDRAND`-derived values flow through a particular execution, not on a fixed property of the binary. When the values affect only data (keys, nonces, ciphertexts), the retired instruction count is identical across recording and replay, and the replay exits cleanly (`exp3-silent`, `mrdrnd`; Intel and AMD). When the values feed a branch decision (carry-flag checks, retry-loop bounds), the instruction count diverges and `rr`’s `validate_regs()` check aborts replay with a FATAL diagnostic (`exp2-intrinsic`). This detection is incidental, a side effect of performance-counter validation rather than a designed safety mechanism.

Crucially, the regime is a per-execution property. Our Rust intrinsic test (`exp-rust` in the artifact), which matches on the `Some(v)/None` returned by `_rdrand64_step`’s carry flag, falls into either regime depending on whether all four `RDRAND` calls succeed in both recording and replay: the common case yields silent divergence, while a single carry-flag failure makes the divergence detectable. A forensic analyst therefore cannot infer from a successful replay that the trace was faithful; the absence of an `rr` diagnostic is necessary but not sufficient for trace fidelity.

5.2 Adversarial Exploitation

A substantial body of security research has established that nondeterministic instructions can be exploited to detect virtualization and instrumentation [20, 47]. Santos-Filho et al. [18] and D’Elia et al. [14] have documented techniques in which adversarial binaries detect dynamic binary instrumentation frameworks through timing discrepancies, memory layout artifacts, and other environmental side channels. While neither study addresses `RDRAND` specifically, their findings establish that adversarial binaries actively probe for analysis environments, and a replay system that fails to faithfully reproduce hardware entropy provides exactly such a detectable artifact. Indeed, detecting `rr` via this gap is trivial: a binary installs a `SIGILL` handler and then executes `RDRAND`. Under `rr`, `CPUID` reports that `RDRAND` is unsupported, yet the instruction succeeds because `rr` cannot suppress it. This inconsistency (a working instruction whose existence `CPUID` denies) is an unambiguous signal that the process is running under replay. Malware that performs this check may alter its behavior to evade forensic examination.

For forensic analysts working with adversarial binaries, the inability to faithfully replay `RDRAND` is a direct threat to analysis integrity: a malware sample that uses `RDRAND` to seed a key, derive a command-and-control address, or gate an anti-analysis check will behave differently during replay, potentially hiding the behavior the analyst seeks to observe.

5.3 Evidence Integrity

In legal and regulatory contexts, the evidentiary value of replay-based forensic analysis depends on the analyst’s ability to demonstrate that the replayed execution is faithful to the original, and silent replay divergence undermines that guarantee. How a given admissibility standard should treat this technical gap is an open question we raise for the digital-evidence community rather than answer here.

Quinn and Alvaro [38] emphasize that correct replay requires identifying and capturing every nondeterministic action of the process. The gap we characterize represents a known, documented class of nondeterministic actions that no existing process-level replay system captures.

6 The Solution Design Space

`RDRAND` and `RDSEED` execute unconditionally at any privilege level, raise no fault, and ignore the `CPUID` feature bit (Section 2.2). `seccomp`-BPF operates on syscall numbers, not instructions, and no MSR or microcode facility analogous to `CPUID` faulting or `UMIP` currently exists. Closing the gap therefore requires intercepting every execution of these instructions by a monitored process and substituting a deterministic value. We identify three properties that any complete solution must satisfy.

Per-process granularity. The mechanism must affect only the monitored process. Solutions that operate at VM granularity, such as KVM’s existing VM-exit support for RDRAND [40, 11], require dedicating an entire virtual machine to a single monitored workload, which is impractical for routine forensic use.

Binary transparency. The mechanism must not require modification, re-compilation, or instrumentation of the monitored binary. Forensic analysis frequently targets stripped, statically linked, or adversarially crafted binaries for which source code is unavailable.

Signal handler robustness. Approaches that deliver a fault (e.g., SIGILL) directly to the process will be detected by binaries that install their own signal handlers, whether for feature probing, anti-analysis, or graceful degradation. The mechanism must interpose before the process’s own handlers observe the fault.

Our goal here is to establish that a solution is architecturally feasible and to identify the most promising path. We survey the principal approaches and their tradeoffs.

6.1 Binary Instrumentation

Dynamic binary instrumentation (Pin [30], DynamoRIO [8]) intercepts every executed instruction, but injects code into the monitored process, violating binary transparency for adversarial targets [20]. `rr` additionally cannot be built on a DBI substrate, since its precise-replay performance counters do not produce consistent counts under instrumentation [32].

6.2 Load-Time Binary Rewriting

Load-time binary rewriting scans executable pages for RDRAND/RDSEED opcodes and patches each site, avoiding the continuous overhead of DBI. It still modifies the process’s code pages (detectable by integrity-checking adversaries [20]) and cannot intercept instructions emitted by JIT compilation or self-modifying code after the initial scan.

6.3 Full-System Emulation

QEMU [6] user-mode emulation intercepts every instruction transparently, but the dynamic translation overhead (up to 10× for PANDA-based malware analysis [16]) is prohibitive for iterative forensic workflows.

6.4 Hypervisor-Based Per-Process Interception

The VM-exit bits for RDRAND and RDSEED exist in the Intel VMX virtualization extensions [24], exposed through KVM [40, 11]. These bits were introduced into mainline Linux as part of the CrossTalk/SRBDS vulnerability mitigations in 2020, which demonstrated that RDRAND results could leak across cores via a shared staging buffer [40]. AMD SVM does not define equivalent VM-exit intercepts for RDRAND or RDSEED [1], so the hypervisor-based approach described

here is currently Intel-specific. The infrastructure to trap these instructions at the hypervisor layer is already present in commodity Linux kernels running on Intel hardware.

This approach satisfies all three properties: the guest’s CR3 identifies the active process at VM-exit time (per-process granularity), interception occurs entirely outside the monitored process’s address space (binary transparency), and the resulting fault can be routed through a `ptrace`-based tracer ahead of the process’s own handlers [32].

The existing KVM infrastructure operates at VM granularity and was designed for security mitigation, not per-process replay [11]. Extending it to support per-process interception with deterministic value substitution, building on the CR3-based process identification, a guest kernel registration interface, and integration with process-level replay machinery, represents a concrete path to closing the gap.

Two observations bound the cost of this extension. First, the underlying trap mechanism is mature: KVM has handled `RDRAND` VM exits in production since 2020 as part of the SRBDS mitigation [11], so the VM-exit handler itself does not need to be designed from scratch; the architectural addition is the per-process gating layer, not the trap. Second, the steady-state overhead is bounded by the recorded program’s `RDRAND` invocation rate, which our prevalence study (Section 4.3) shows is structurally low: only 99 of 33,909 (0.3%) scanned ELF binaries contain `RDRAND` or `RDSEED` at all, and the binaries that do are crypto libraries that invoke the instruction once or a small constant number of times to seed a DRBG. Granting a loose upper bound of 100 invocations per recorded process at 10^4 added cycles per VM exit yields ~ 1 ms of added per-process recording time; the $\sim 99.7\%$ of processes that never execute these instructions incur zero overhead from the extension.

6.5 Hardware Per-Process Trapping

The most elegant long-term solution would be a hardware mechanism for per-process `RDRAND` trapping, analogous to per-process `CPUID` faulting [31]. The CrossTalk/SRBDS discussion noted this as a desirable capability, but no such mechanism has been implemented [11]. The piecemeal accretion of per-instruction interception mechanisms (UMIP, `CPUID` faulting, `CR4.TSD`, and now the need for `RDRAND` trapping) suggests that a unified architectural facility for per-process unprivileged instruction interception would be preferable to continued ad hoc additions.

7 Related Work

7.1 Record-and-Replay Systems

The intellectual lineage of execution replay stretches back to LeBlanc and Mellor-Crummey’s Instant Replay in 1987 [28], which established that replay systems

Table 5. Treatment of `RDRAND`/`RDSEED` across replay systems.

System	Logging boundary	Covered?
<code>rr</code> [32]	syscall + dyn-lib patching	Partial
Scribe [27]	syscall	No
WEBRR [3]	Blink/V8 web-API shims	Above boundary
PinPlay [34]	every instruction (Pin)	Yes
TTD (Nirvana) [7]	every instruction (DBT)	Yes
PANDA [16]	QEMU TCG translation	Yes
ReVirt [17], Aftersight [10]	VM I/O	Yes (in principle)
REPT [13]	branch decisions (IPT)	No (re-issued)

need only capture what is nondeterministic. Subsequent taxonomic work [19] organized the space along dimensions including abstraction level and types of nondeterminism.

VM-level systems including ReVirt [17], Aftersight [10], ExecRecorder [33], and PANDA [16] achieve completeness by logging all VM inputs. Emulation-based systems could intercept `RDRAND` during translation; whether any deployed system currently does so is an open question.

Process-level systems including `rr` [32], Scribe [27], and PinPlay [34] narrow the boundary to a single process. `rr` handles `RDRAND` only narrowly, via cooperative `CPUID` suppression and opcode patching of one known `libstdc++` call site [32]. Scribe intercepts nondeterminism via system call interposition and does not handle instructions that complete in user space. PinPlay could intercept `RDRAND` via Pin [30], and Microsoft’s Time Travel Debugging similarly captures it via the Nirvana/iDNA engine [7], but both rely on in-process code injection that is detectable by adversarial binaries [20] (Section 6). CRIU [12] performs checkpoint/restore rather than execution-trace replay; Intel Processor Trace-based replay [13] records only branch decisions; both re-issue `RDRAND` on reconstruction and obtain fresh values.

More recent work includes OmniTable [39], which builds SQL-queryable debugging on top of `rr` traces and therefore inherits `rr`’s coverage, and WEBRR [3], which records at Chromium’s Blink/V8 web-API boundary via instrumentation shims rather than at the syscall boundary. WEBRR is therefore unaffected by the gap for the JS-observable randomness in its threat model (`Math.random`, `Date`, `crypto.getRandomValues`, network responses): all such values are shimmed above the instruction layer where `RDRAND` lives.

Table 5 consolidates these treatments: coverage tracks the logging boundary rather than the tool, so every syscall-boundary system inherits the gap and every instruction-level system (DBI, DBT, hypervisor-assisted translation) covers `RDRAND` as a side effect. Closing the gap without abandoning the syscall-boundary architecture is the design problem of Section 6.

7.2 Hypervisor-Based Instruction Interception

Popek and Goldberg [37] established the formal conditions for virtualizable architectures, and Robin and Irvine [41] identified 17 sensitive but unprivileged x86 instructions that could not be cleanly virtualized. `RDRAND` fits this pattern exactly: it exposes state without trapping.

Xen [5] addressed the x86 virtualization gap through paravirtualization and intercepts sensitive unprivileged instructions via hypercalls. KVM [25] leverages hardware virtualization extensions for selective instruction interception via VM exits. Neither enables per-process `RDRAND` interception.

Ether [15] and DRAKVUF [29] demonstrate that hypervisors can transparently observe guest execution for malware analysis, but neither performs selective interception of individual instructions via VM-exit control bits, the mechanism required for `RDRAND` interception.

7.3 VM Detection and Adversarial Instructions

Garfinkel et al. [20] provide a thorough treatment of why building a truly transparent VMM is fundamentally infeasible. Earlier work [47] systematically enumerates the complete set of unprivileged instructions that expose execution environment state and establishes them as actively-used VM-detection primitives, including `RDRAND`. Meeting the bar set by anti-VM detection techniques therefore requires that emulated `RDRAND` responses be architecturally indistinguishable from native hardware.

8 Conclusion

Hardware entropy instructions complete unconditionally at any privilege level, leaving process-level record-and-replay systems that log at the syscall boundary structurally blind to their values (Sections 4, 7). The gap manifests through library-internal `RDRAND`, compile-time emission via `-mrdrnd` or `-march=native`, and the eight-year absence of AMD CPUID faulting; the resulting divergence is silent whenever the `RDRAND` return values affect only data (Section 5). Hypervisor-based per-process interception is the most promising near-term path on Intel: the KVM VM-exit machinery introduced for SRBDS [40] already traps these instructions, so the architectural addition is the per-process gating layer rather than the trap itself. AMD SVM lacks equivalent intercepts, and non-x86 ISAs exposing analogous instructions (Section 2.2) will require their own mechanisms. Until the gap is closed, the reliability of replay-based forensic conclusions for any binary that may execute `RDRAND`—directly or through a linked library—cannot be guaranteed.

Acknowledgments. We thank the anonymous WSDf reviewers for their constructive feedback.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Advanced Micro Devices, Inc.: AMD64 Architecture Programmer’s Manual (2024), <https://www.amd.com/system/files/TechDocs/40332.pdf>
2. Advanced Micro Devices, Inc.: RDSEED Failure on AMD “Zen 5” Processors (2025), <https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7055.html>
3. Allen, J., Yang, Z., Xiao, F., Landen, M., Perdisci, R., Lee, W.: WEBRR: A forensic system for replaying and investigating web-based attacks in the modern web. In: Proceedings of the 33rd USENIX Conference on Security Symposium. pp. 1669–1686. SEC ’24, USENIX Association (2024)
4. Arm Limited: Arm Architecture Reference Manual for A-profile Architecture (2024), <https://developer.arm.com/documentation/ddi0487/latest/>, aRM DDI 0487; FEAT_RNG (RNDR/RNDRS) introduced in ARMv8.5-A
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. **37**(5), 164–177 (2003). <https://doi.org/10.1145/1165389.945462>
6. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 41. ATEC ’05, USENIX Association (2005). <https://doi.org/10.5555/1247360.1247401>
7. Bhansali, S., Chen, W.K., de Jong, S., Edwards, A., Murray, R., Drinic, M., Mihocka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: Proceedings of the 2nd International Conference on Virtual Execution Environments. pp. 154–163. VEE ’06, Association for Computing Machinery (2006)
8. Bruening, D.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. thesis, Massachusetts Institute of Technology (2004)
9. Chen, Y., Zhang, S., Guo, Q., Li, L., Wu, R., Chen, T.: Deterministic Replay: A Survey. ACM Computing Surveys **48**(2), 1–47 (Nov 2015). <https://doi.org/10.1145/2790077>
10. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling dynamic program analysis from execution in virtual environments. In: USENIX 2008 Annual Technical Conference. pp. 1–14. ATC’08, USENIX Association (2008)
11. Corbet, J.: The “special register buffer data sampling” hardware vulnerability (2020), <https://lwn.net/Articles/822595/>
12. CRIU Contributors: CRIU: Checkpoint/Restore In Userspace. Project website (2025), <https://criu.org/>
13. Cui, W., Ge, X., Kasikci, B., Niu, B., Sharma, U., Wang, R., Yun, I.: REPT: Reverse Debugging of Failures in Deployed Software. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 17–32. USENIX Association (2018), <https://www.usenix.org/conference/osdi18/presentation/weidong>
14. D’Elia, D.C., Invidia, L., Palmaro, F., Querzoni, L.: Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. Digital Threats: Research and Practice **3**(2) (2022). <https://doi.org/10.1145/3478520>

15. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. pp. 51–62. CCS '08, Association for Computing Machinery (2008). <https://doi.org/10.1145/1455770.1455779>
16. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable Reverse Engineering with PANDA. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop. pp. 1–11. PPREW-5, Association for Computing Machinery (2015). <https://doi.org/10.1145/2843859.2843867>
17. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M.A., Chen, P.M.: ReVirt: Enabling intrusion analysis through Virtual-Machine logging and replay. In: 5th Symposium on Operating Systems Design and Implementation (OSDI 02). USENIX Association (2002)
18. Filho, A.S., Rodríguez, R.J., Feitosa, E.L.: Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks. *Digital Threats: Research and Practice* **3**(2) (2022). <https://doi.org/10.1145/3480463>
19. Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, Bosschere, K.D.: A taxonomy of execution replay systems. In: Intern. Conf. on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet (2003)
20. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: VMM detection myths and realities. In: Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems. pp. 1–6. HOTOS'07, USENIX Association (2007). <https://doi.org/10.5555/1361397.1361403>
21. GCC incorrectly assumes all CPUs where `-march=native` resolves to `-march=bdver4` will have RDRND support (2024), https://gcc.gnu.org/bugzilla/show_bug.cgi?id=116854
22. GCC Project: Libstdc++ random.cc — std::random_device Implementation (2024), <https://github.com/gcc-mirror/gcc/blob/releases/gcc-14.2.0/libstdc++-v3/src/c++11/random.cc>
23. Guo, K.F., Huey, K.: Segmentation fault with std::random_device (rr 5.5.0 and master) (2022), <https://github.com/rr-debugger/rr/issues/3304>
24. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual (2025), <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
25. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: Kvm: The linux virtual machine monitor. In: Proceedings of the Linux Symposium. pp. 225–230 (2007)
26. Kuechler, A., Mantovani, A., Chen, Y., Bilge, L., Balzarotti, D.: Does every second count? Time-based evolution of malware behavior in sandboxes. In: Proceedings of the Network and Distributed System Security Symposium (NDSS 2021). Internet Society (2021). <https://doi.org/10.14722/ndss.2021.24475>
27. Laadan, O., Viennot, N., Nieh, J.: Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. pp. 155–166. SIGMETRICS '10, Association for Computing Machinery (2010). <https://doi.org/10.1145/1811039.1811057>
28. Leblanc, Mellor-Crummey: Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers* **C-36**(4), 471–482 (1987). <https://doi.org/10.1109/TC.1987.1676929>
29. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis sys-

- tem. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 386–395. ACSAC '14, Association for Computing Machinery (2014). <https://doi.org/10.1145/2664243.2664252>
30. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05). pp. 190–200. ACM (2005). <https://doi.org/10.1145/1065010.1065034>
 31. X86/arch_prctl: Add ARCH_[GET|SET]_CPUID for controlling the CPUID instruction (2017), <https://lwn.net/Articles/713970/>
 32. O’Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., Partush, N.: Engineering record and replay for deployability. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. pp. 377–389. USENIX ATC '17, USENIX Association, USA (Jul 2017)
 33. de Oliveira, D.A.S., Crandall, J.R., Wassermann, G., Wu, S.F., Su, Z., Chong, F.T.: ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. pp. 66–71. ASID '06, Association for Computing Machinery (2006). <https://doi.org/10.1145/1181309.1181320>
 34. Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J.: PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 2–11. CGO '10, Association for Computing Machinery (2010). <https://doi.org/10.1145/1772954.1772958>
 35. Petkov, B.: [patch] x86/CPU/AMD: Add CPUID faulting support (2025), <https://lore.kernel.org/all/20250528213105.1149-1-bp@kernel.org/>, linux kernel mailing list, 2025-05-28
 36. Petkov, B.: x86/CPU/AMD: Add CPUID faulting support (2025), <https://github.com/torvalds/linux/commit/65f55a30176662ee37fe18b47430ee30b57bfc98>, linux kernel commit 65f55a3017, merged 2025-06-21
 37. Popek, G.J., Goldberg, R.P.: Formal requirements for virtualizable third generation architectures. Commun. ACM **17**(7), 412–421 (1974). <https://doi.org/10.1145/361011.361073>
 38. Quinn, A., Alvaro, P.: Deterministic Record-and-Replay. Communications of the ACM **68**(5), 32–34 (Apr 2025). <https://doi.org/10.1145/3724381>
 39. Quinn, A., Flinn, J., Cafarella, M., Kasikci, B.: Debugging the OmniTable way. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 357–373. USENIX Association (2022), <https://www.usenix.org/conference/osdi22/presentation/quinn>
 40. Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C.: CrossTalk: Speculative Data Leaks Across Cores Are Real. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1852–1867 (2021). <https://doi.org/10.1109/SP40001.2021.00020>
 41. Robin, J.S., Irvine, C.E.: Analysis of the Intel Pentium’s ability to support a secure virtual machine monitor. In: Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9. p. 10. SSYM'00, USENIX Association (2000)
 42. CPUID faulting required to disable CPUID features (2024), <https://github.com/rr-debugger/rr/issues/3799>
 43. rr-debugger project: Zen (2024), <https://github.com/rr-debugger/rr/wiki/Zen>, rr wiki page documenting AMD Zen support and known issues
 44. Systemd Project: NEWS: Release notes for systemd (2026), <https://github.com/systemd/systemd/blob/359640e/NEWS>

45. The Rust Project Developers: Getrandom RDRAND Backend (2024), <https://github.com/rust-random/getrandom/blob/v0.3.2/src/backends/rdrand.rs>
46. Wakely, J.: PR libstdc++/85494 use rdseed and rand_s in std::random_device (2019), <https://gcc.gnu.org/git/?p=gcc.git;a=commit;h=b0c0d878a8b5bf39dbea4c192fed26d340524439>, gCC commit b0c0d878, 2019-05-29; released in GCC 10 (libstdc++.so.6.0.28). Replaces the prior `ebx == signature_INTEL_ebx` guard with `(ebx == signature_INTEL_ebx || ebx == signature_AMD_ebx)`, extending the `std::random_device` RDRAND/RDSEED dispatch to AMD CPUs for the first time.
47. Wright, J.L.: Where am I? Operating System and Virtualization Identification Without System Calls. In: Cyber Security Symposium (2017)