

Entropy Sources in Confidential Cloud Environments: Revisiting the BSI 2016 Assessment of RDRAND and RDSEED under Intel TDX and AMD SEV-SNP

Alina Abraham
*studying Angewandte Informatik
at Hochschule Flensburg
Flensburg, Germany
alina.abraham@stud.hs-flensburg.de*

Ben Lubowski
*studying Angewandte Informatik
at Hochschule Flensburg
Flensburg, Germany
ben.lubowski@stud.hs-flensburg.de*

Version 1.0 · 20 September 2026

Abstract—The BSI 2016 study on random number generation in virtual environments established that a Linux guest can obtain sufficient entropy under appropriate configuration, while identifying boot-time entropy starvation, noise source degradation under VMM operation, and a demonstrated interception of RDRAND by a malicious hypervisor. Because that study assumed a non-malicious VMM throughout, its conclusions do not reach the adversarial cloud deployments that motivate the present work. This paper revisits them on current Intel and AMD server hardware and on a Linux kernel whose entropy architecture has since been redesigned rather than refined, and carries them forward into hardware-enforced confidential computing. We assess the min-entropy of RDRAND and RDSEED under NIST SP 800-90B on bare metal on both vendor platforms and in an AMD SEV-SNP guest, we re-examine whether the interception attack demonstrated in 2016 remains feasible on current VMM implementations, using a KVM guest for that part, we establish from the specifications and from hypervisor and firmware source what Intel TDX changes architecturally, and we trace what becomes of the Linux combiner guarantee once the hypervisor is excluded from the Trusted Computing Base. Across fourteen datasets the delivered output is indistinguishable from full entropy, on both vendors and inside the confidential guest alike, and a ladder of constructed sources shows how little that establishes, since a generator carrying thirty-two bits of entropy receives the same assessment to six decimal places, under the NIST estimators and under the AIS 31 suite and under a compression check. The interception demonstrated in 2016 reproduces on current hypervisor code, and the architectures remove it on the instruction path while leaving the host its say over what a guest is told about the instruction. The resulting picture is one of narrowing rather than repair. Of the eleven entropy sources feeding the input pool, confidential computing alters the assessment of exactly one, so that the security of a confidential guest’s entropy pool comes to rest on the single proposition that the CPU instruction delivers the entropy its specification claims, which no party outside the manufacturer is in a position to verify and which no assessment of output can confirm.

Index Terms—Entropy, Random number generation, RDRAND, Confidential computing, Intel TDX, AMD SEV-SNP, NIST SP 800-90B, BSI AIS 20/31

CONTENTS

1	Introduction	4
1.1	Motivation	4
1.2	Problem Statement	5
1.3	Research Questions	5
1.4	Scope and Constraints	5
1.5	Structure of the Paper	5
2	Background and Terminology	5
2.1	Entropy and Information Theory	5
2.1.1	Shannon Entropy	6
2.1.2	Min-Entropy and Its Relevance for Cryptography	6
2.1.3	IID and Non-IID Sources	7
2.2	Random Number Generator Architecture	8
2.2.1	The Noise Source, Conditioning and DRBG Pipeline	8
2.2.2	A Taxonomy of TRNG, CSPRNG and DRBG	9
2.2.3	The XOR Combiner Approach and the XOR Hash Lemma	9
2.3	Relevant Standards	10
2.3.1	Regulatory Framework	11
2.3.2	NIST SP 800-90A and DRBG Construction	12

2.3.3	NIST SP 800-90B and Entropy Estimation	12
2.3.4	BSI AIS 20/31	13
3	Architecture of Noise Sources	14
3.1	General Noise Source Architecture	14
3.2	The Current Linux Random Number Generator	16
3.2.1	/dev/random and /dev/urandom in Current Kernels	16
3.2.2	How Linux Mixes Entropy Sources	16
3.2.3	Entropy Sources Contributing to the Input Pool	17
3.2.4	RDRAND as One Input to the Entropy Pool	19
3.3	Intel RDRAND and RDSEED	20
3.3.1	Internal Architecture of the Intel DRNG	20
3.3.2	Differences and Use Cases of RDRAND and RDSEED	22
3.3.3	Vendor Opacity and the Limits of Auditability	22
3.4	AMD RDRAND and RDSEED	23
3.4.1	The Publicly Documented AMD Architecture	23
3.4.2	Differences to the Intel Implementation	24
4	Virtual Machine Monitor Impact on Noise Sources	25
4.1	KVM/QEMU in Cloud Virtualization Architecture	25
4.2	How VMMs Intercept CPU Instructions	26
4.2.1	VM-Exit Behavior on Intel and AMD	26
4.2.2	Full Instruction Emulation and Binary Rewriting	27
4.3	VMM Impact on RDRAND and RDSEED	27
4.3.1	Reproducing the BSI 2016 Interception Attack	27
4.3.2	VM-Exit Configuration on Intel	28
4.3.3	VM-Exit Behavior on AMD	29
4.4	VMM Impact on the Linux Entropy Pool	30
4.4.1	Entropy Sources in Virtualized Environments	30
4.4.2	Boot-Time Entropy Starvation	31
4.4.3	The Combiner under VMM Interference	33
5	Confidential Computing as a Trust Model Shift	34
5.1	Eliminating the Trusted VMM Assumption	34
5.2	Intel TDX	34
5.2.1	Architecture and Trusted Computing Base	34
5.2.2	What TDX Does and Does Not Protect	35
5.3	AMD SEV-SNP	36
5.3.1	Architecture and Comparison to TDX	36
5.3.2	What SEV-SNP Does and Does Not Protect	37
5.4	Remote Attestation of CPU Type and Enclave State	38
5.5	Entropy Sources under Confidential Computing	39
5.5.1	Reassessment of Entropy Sources in TDX and SEV-SNP Guests	39
5.5.2	RDRAND and RDSEED as the Mandated Primary Source	41
5.5.3	The Combiner Argument under Confidential Computing	42
5.6	Remaining Trust Gaps under Confidential Computing	43
6	Entropy Assessment Methodology	45
6.1	Entropy Assessment Frameworks	45
6.2	Experimental Platform	47
6.2.1	Access Levels and What Each Permits	47
6.2.2	Hardware Platforms	48
6.2.3	Environments and the Asymmetry Between Them	49
6.2.4	Platform Configuration and Its Defaults	49
6.2.5	Provenance Through Manifests and Attestation	51
6.2.6	Analysis Pipeline	52
6.3	Measurement Families	53
6.3.1	M1 Baseline Min-Entropy Across Environments	53
6.3.2	M2 Detectability of a Known-Defective Source	53
6.3.3	M3 Instruction Interception under Confidential Computing	53
6.3.4	M4 RDSEED under Contention	54
6.3.5	M5 Time to Fully Seeded	54
6.3.6	M6 Restart Tests	54
6.4	What the Results Are Compared Against	55
6.5	What This Methodology Cannot Establish	55
7	Results	56
7.1	Baseline on Bare Metal	56
7.2	What the Assessment Detects and What It Does Not	57
7.3	Interception under KVM/QEMU	58
7.4	The Source under Contention	59
7.5	Time to a Seeded Pool	59
7.6	Behaviour Across Restarts	59
7.7	Measurements under Confidential Computing	60
7.8	Measured Output Against Vendor Certification	60
8	Discussion	60
8.1	Sufficiency of RDRAND Without an External HSM	61
8.2	The Combiner as a Mitigation	61
8.3	Conditions for Trusting RDRAND and RDSEED in Cloud Deployments	62
8.4	Implications for BSI High-Security Requirements	62

9 Conclusion	62
9.1 Summary of Findings	62
9.2 Comparison to BSI 2016	63
9.3 Recommendations for Practitioners	63
9.4 Future Work	63
Appendix A: Checklist for RDRAND Assessment in Cloud Environments	64
Appendix B: Test Environment Specifications	64
Appendix C: Raw Measurement Data	65
Appendix D: References	67

1 INTRODUCTION

1.1 Motivation

Cryptographic security is fundamentally dependent on the availability of high-quality random numbers. Keys, nonces, and initialization vectors must be unpredictable to any adversary, which means formally that they must be generated with sufficient entropy. When the underlying random number generation is flawed, every cryptographic primitive built on top of it is compromised, regardless of its mathematical soundness. Practice has demonstrated this repeatedly, as when the 2008 Debian OpenSSL vulnerability reduced the effective key space to approximately 32,000 values through a corrupted entropy pool [1], or when the 2010 Sony PlayStation 3 exploit achieved full private key recovery because a constant nonce had been reused in ECDSA signatures [2]. In both cases the cryptographic algorithm was sound while the random number generator was not.

In cloud environments, entropy generation is structurally more difficult than on physical hardware, because virtual machines are isolated from the hardware event sources that traditionally contribute to an operating system’s entropy pool, so that keyboard activity, disk interrupt timing, and other peripheral events are either absent altogether or mediated by the hypervisor.

This isolation creates two distinct problems. The first is that the entropy pool accumulates randomness more slowly, which leads to entropy starvation during the boot phase, at the very moment when cryptographic daemons initialize and generate keys before sufficient entropy has been collected. The BSI study on random number generation in virtual environments, conducted in 2016, demonstrated this concretely by measuring a KVM-based system on which the Linux entropy pool required between 25 and 90 seconds to accumulate 128 bits of entropy after boot, while the operating system itself was fully initialized within two seconds [3]. OpenSSH generates its host keys during precisely this window, with potentially only a few bits of entropy available.

The second problem is that the hypervisor sits between the guest operating system and the hardware, so that a compromised or malicious hypervisor can in principle observe or manipulate the entropy sources available to a guest, undermining the integrity of any randomness generated within it.

Intel RDRAND and RDSEED are CPU instructions that provide hardware-generated random numbers directly from an on-chip entropy source, bypassing the operating system entropy pool entirely. Because they operate at the level of the instruction set, they are accessible from within a guest without hypervisor mediation, at least in principle, which makes them an attractive answer to the cloud entropy problem, being a hardware entropy source that depends on no peripheral events, does not suffer from entropy starvation, and cannot be withheld by the hypervisor.

The picture is more complex than that, however. The BSI 2016 study identified that a VMM can intercept the RDRAND

instruction issued by a guest, because Intel’s architecture explicitly allows VM-exits to be configured for it, and one of the study’s authors demonstrated the point with a proof-of-concept implementation in KVM in which ten lines of C code sufficed to intercept every RDRAND call from a guest and return zero in place of a random value [3]. RDRAND’s internal implementation is moreover not independently auditable, since the physical entropy source and the conditioning pipeline are entirely contained within the CPU silicon and no third-party validation comparable to that of a FIPS-certified HSM is available. The Snowden revelations of 2013, which exposed the NSA’s deliberate introduction of a backdoor into the NIST-standardized Dual EC DRBG algorithm, raised the broader concern that hardware entropy sources from US chip manufacturers may not be fully trustworthy [4].

Two significant developments have changed the landscape since 2016. The first is hardware-enforced confidential computing, introduced by Intel Trust Domain Extensions (TDX) and AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP), under which the hypervisor is architecturally excluded from the Trusted Computing Base while guest memory and CPU state are protected from hypervisor access. In this model the BSI 2016 interception attack is no longer possible by design, because the instruction set architecture of a Trust Domain places RDRAND and RDSEED outside the set of instructions whose execution the hypervisor may trap. The TDX Module Base Architecture Specification enumerates the instructions a Trust Domain cannot execute or that are reflected to the guest, and neither RDRAND nor RDSEED appears in that enumeration [5, Sec. 11.7], while the published source of the TDX Module fixes both exiting controls to zero and places them outside the mask of bits a host may vary [6]. The Linux TDX guest kernel security specification acknowledges the consequence explicitly, classifying interrupt timing as controlled by the untrusted hypervisor inside a TDX guest and therefore unsuitable as an entropy source, and mandating RDRAND as the primary entropy source during boot [7].

The second development is that modern Linux kernels have adopted a combiner approach to entropy collection, in which multiple sources including RDRAND are mixed into the entropy pool through a cryptographic combiner. The XOR Hash Lemma guarantees that the combined output remains cryptographically sound as long as at least one source is uncompromised and independent of the others, which provides a degree of resilience that the architecture examined in 2016 did not have [8].

These developments raise a question that the BSI 2016 study did not and could not address. Does the combination of RDRAND’s hardware entropy, modern Linux entropy pooling, and confidential computing isolation provide sufficient cryptographic randomness for high-security cloud deployments without requiring an external hardware security module?

1.2 Problem Statement

The BSI 2016 study established that Linux running in a virtual machine can obtain sufficient entropy under appropriate configuration, but identified significant vulnerabilities: entropy starvation at boot, noise source degradation under VMM operation, and the theoretical and demonstrated possibility of RDRAND interception by a malicious hypervisor. The study explicitly assumed a non-malicious VMM throughout, a constraint that limits its applicability to adversarial cloud environments.

Since 2016, CPU architectures, Linux kernel entropy mechanisms, and cloud deployment models have changed substantially, so that the study’s findings were derived from hardware and software now a full generation behind, given that the CPUs tested predate confidential computing extensions, that the Linux kernel has since replaced its entropy architecture, and that cloud computing has shifted from general-purpose virtualization towards increasingly security-critical workloads including AI training and inference on sensitive data.

This paper revisits the BSI 2016 findings with current hardware and software. We reproduce the study’s core experimental methodology (entropy measurement of RDRAND and RDSEED using NIST SP 800-90B) on modern Intel and AMD server CPUs, on bare metal, in a KVM guest, and in an AMD SEV-SNP guest. We evaluate whether the interception attack demonstrated in 2016 remains feasible on current VMM implementations, and we establish from the specifications and from the hypervisor and firmware source code what Intel TDX and AMD SEV-SNP change about it architecturally. We assess how the Linux combiner approach changes the security model, and whether the resulting system can satisfy BSI high-security requirements without an external HSM.

1.3 Research Questions

This paper addresses the following research questions:

RQ1 (Entropy quality). What can an SP 800-90B assessment of RDRAND and RDSEED output establish about the entropy that output contains, and can such an assessment distinguish it from a source known to lack entropy?

RQ2 (Interception feasibility). Is the RDRAND/RDSEED interception attack demonstrated by the BSI 2016 study still feasible on current KVM/QEMU implementations and current Intel and AMD CPUs? Does the VM-exit behavior differ between Intel and AMD?

RQ3 (Confidential computing as a resolution). To what extent do Intel TDX and AMD SEV-SNP remove the hypervisor’s influence over RDRAND and RDSEED, and on what does the entropy supply of a confidential guest then rest?

RQ4 (Sufficiency for high-security deployments). Under what conditions and architectural constraints can RDRAND and RDSEED, combined with the Linux entropy pool, satisfy BSI high-security requirements for cloud-based systems including those where a physical TPM or HSM is unavailable?

1.4 Scope and Constraints

This study focuses exclusively on server-side cloud infrastructure. Desktop virtualization, embedded systems, and consumer hardware are outside the scope. The motivating deployment context is large-scale cloud and AI infrastructure, where confidential computing is increasingly relevant for protecting sensitive workloads and model data from the cloud provider itself.

On the software side, the analysis is limited to Linux. Because the Linux kernel’s entropy architecture is well documented, actively maintained, and dominant across cloud server deployments, it is the natural successor to the system analyzed in the BSI 2016 study. The specific VMM under investigation is KVM/QEMU, which underlies most major cloud provider infrastructure.

On the hardware side, both Intel and AMD server CPUs are considered. The BSI 2016 study noted that AMD’s RDRAND implementation did not appear to support VM-exit configuration in contrast to Intel’s, a difference with direct implications for the interception attack [3]. This distinction is examined against the current specification and the KVM source.

The study does not assess the trustworthiness of Intel’s or AMD’s internal entropy source design beyond what can be measured externally via NIST SP 800-90B. Questions of hardware backdoors, while relevant context, are outside the empirical scope of this work.

1.5 Structure of the Paper

The remainder of this paper is structured as follows. Chapter 2 establishes the theoretical foundations, covering entropy and information theory, random number generator architecture and taxonomy, and the relevant standards, after which Chapter 3 describes the architecture of the noise sources under investigation, including the current Linux random number generator and the Intel and AMD RDRAND and RDSEED implementations. Chapter 4 analyzes the impact of virtual machine monitors on those noise sources and reproduces the BSI 2016 interception experiment on current hardware, while Chapter 5 examines confidential computing as a structural shift in the trust model, covering Intel TDX and AMD SEV-SNP. The empirical part of the work begins with Chapter 6, which describes the entropy assessment methodology, and continues in Chapter 7 with the experimental results. Chapter 8 discusses those results in the context of the research questions, and Chapter 9 concludes with recommendations and directions for future work.

2 BACKGROUND AND TERMINOLOGY

2.1 Entropy and Information Theory

Entropy is a formal measure used to quantify the unpredictability of a random variable’s outcomes, and thus the standard criterion for evaluating whether a random number generator meets cryptographic adequacy. Intuitively, entropy

captures the level of uncertainty or surprise present when observing the result of a random process.

When entropy is low, one outcome is significantly more probable than the others, making the result largely predictable and therefore each observation carries little new information. When entropy is high, outcomes are more equally distributed, making each draw harder to anticipate. In the extreme case of a uniform distribution, no outcome is more likely than any other, meaning every observation is maximally surprising, and entropy reaches its theoretical peak.

Since random numbers are only as secure as the unpredictability of their source, a precise and measurable definition of that unpredictability is necessary. The following subsections define this concept formally, starting with Shannon entropy as the classical measure, then motivating min-entropy as the appropriate criterion for cryptographic use, and finally establishing the IID vs. Non-IID distinction that determines how entropy is correctly estimated in practice.

2.1.1 Shannon Entropy

A clear picture of Shannon entropy begins with the notion of surprise, which can be understood as the information content of a single outcome, so that the more surprising an outcome is, the more information observing it conveys. Formally, the surprise of an outcome x is defined as

$$I(x) = -\log_b P(x) \quad (1)$$

from which it follows that surprise is inversely related to probability, the less likely an outcome being the more surprising and therefore the more informative it is to observe. Since a random process produces different outcomes with different probabilities, the overall uncertainty of the source is best described not by the surprise of any single outcome but by the expected surprise across all possible outcomes, which Shannon formalizes as a probability-weighted **average of surprise** [9]:

$$H(X) = -\sum_{x \in X} P(x) \log_b P(x) \quad (2)$$

where X is a discrete random variable with possible outcomes x . Setting $b=2$ gives the entropy in **bits** and $b=e$ in **nats** (natural units), and by the convention that $0 \log 0 = 0$, outcomes with zero probability contribute nothing to the sum.

Consider a fair coin with $P(\text{Head}) = P(\text{Tail}) = 0.5$, where each outcome carries a surprise of $-\log_2 0.5 = 1$ bit, so that the expected surprise across both outcomes is also exactly one bit. Each coin flip therefore conveys exactly one bit of new information, which is the maximum possible for a two-outcome process and is achieved precisely because neither outcome is more predictable than the other.

Shannon entropy satisfies several properties that matter for what follows. It is **non-negative**, with $H(X) \geq 0$ and equality holding only where the outcome is deterministic, meaning that one outcome has probability one while all others have probability zero, so that no uncertainty remains. It is furthermore **maximized by a uniform distribution**, so that a variable with n possible outcomes satisfies $H(X) \leq \log_b n$ with equality

when all outcomes are equally likely, which confirms the intuition established above that the more evenly spread a distribution is, the higher its uncertainty and the more information each observation carries.

Two further properties become relevant when analyzing entropy sources that exhibit dependencies or that an adversary partially knows. The **chain rule** states that the joint entropy of two random variables X and Y decomposes as $H(X, Y) = H(X) + H(Y|X)$, where the **conditional entropy** $H(Y|X)$ measures the uncertainty remaining in Y once X is known and is itself given by $H(Y|X) = -\sum_x P(x) H(Y|X=x)$. This becomes directly relevant in Chapter 4, where the question is how much entropy an RNG output retains once an attacker holds partial knowledge of the system state, such as the approximate boot time of a virtual machine. In the continuous case entropy generalizes to **differential entropy** $h(X) = -\int p(x) \log p(x) dx$, which can be negative and does not share all the properties of its discrete counterpart, although since every entropy source analyzed in this paper is discrete it is not considered further.

Useful as it is as a measure of average uncertainty, Shannon entropy has a fundamental limitation for cryptographic purposes, in that it describes the source as a whole by averaging surprise across all outcomes, whereas a rational adversary does not attack on average but identifies the single most probable outcome and exploits it directly. A distribution in which one outcome occurs with probability 0.99 still carries non-trivial Shannon entropy, yet an attacker who always guesses that outcome succeeds ninety-nine times in a hundred. Shannon entropy therefore fails to capture the worst-case predictability that cryptographic security actually requires, and it is this gap that motivates min-entropy as the appropriate measure, introduced in the following subsection.

2.1.2 Min-Entropy and Its Relevance for Cryptography

Where Shannon entropy measures the average unpredictability of a source, cryptographic security demands a stricter guarantee, namely a measure that captures the best an adversary can do in a single guess rather than what they achieve on average. Min-entropy provides exactly that and is defined as [10]:

$$H_\infty(X) = -\log_2 \left(\max_x P(x) \right) \quad (3)$$

Rather than averaging over all outcomes, min-entropy isolates the single most probable outcome and measures only its predictability. The adversary's optimal success probability in one guess is directly given by:

$$P(\text{success}) = 2^{-H_\infty(X)} \quad (4)$$

This makes min-entropy an immediate and operational security measure, since it states exactly how likely the best possible attack is to succeed.

Returning to the example above, with $p_{\max} = 0.99$:

$$H_\infty(X) = -\log_2(0.99) \approx 0.014 \text{ bits} \quad (5)$$

Nearly zero correctly reflects that the source is almost entirely predictable, regardless of what Shannon entropy reports.

At the other extreme, consider a source uniform over 2^{128} values, where every outcome has probability 2^{-128} :

$$H_{\infty}(X) = -\log_2(2^{-128}) = 128 \text{ bits} \quad (6)$$

The adversary's success probability in one guess is 2^{-128} , which is computationally negligible. This is the target for cryptographic key generation.

Min-entropy is always a lower bound on Shannon entropy:

$$H_{\infty}(X) \leq H(X) \quad (7)$$

with equality if and only if X is uniformly distributed, which means that Shannon entropy systematically overestimates the security of any non-uniform source. A uniform distribution is the cryptographic ideal precisely because it is the one distribution for which the two measures agree, the average case and the worst case coinciding there. Since uniformity cannot be assumed in advance but must instead be verified through measurement, min-entropy is the appropriate metric, and an argument that measured Shannon entropy and concluded from it that a source is secure would be unsound.

In practice the security parameter 2^n of a cryptographic system defines the promised key space, which is to say how many keys could exist, while min-entropy measures the actual key space, meaning how much of that space the random number generator reliably reaches. A system claiming 128-bit security must therefore draw on an entropy source delivering $H_{\infty} \geq 128$ bits, and where the source is biased so that the effective min-entropy amounts to only 40 bits, the attacker's search space collapses from 2^{128} to 2^{40} regardless of how strong the cryptographic algorithm is.

Output carrying close to 1.0 bits of min-entropy per bit corresponds to a near-uniform distribution and can be consumed directly, one bit of output for one bit of required entropy. A source delivering less per bit is not thereby unusable, since the shortfall can be compensated by collecting proportionally more output before the required amount has accumulated, so that what ultimately matters is whether the total entropy drawn reaches the security level being claimed rather than the rate at which any single output bit supplies it. This distinction becomes concrete for the AMD implementation examined in Section 3.4.1, whose certified per-block entropy is low enough that a correctly implemented consumer must accumulate across many calls.

NIST SP 800-90B, the standard used to assess entropy sources in this paper, always reports min-entropy as a lower bound rather than an exact value. A result of 0.98 bits per bit means that at least 0.98 bits of min-entropy are guaranteed while the true value may well be higher, a conservative convention adopted deliberately because in security contexts underestimating entropy is preferable to overestimating it [11].

Attackers moreover rarely operate without any knowledge of the system, and side information, by which is meant observable facts about the environment, can substantially reduce the effective unpredictability of an entropy source. Conditional min-entropy formalizes this as

$$H_{\infty}(X | Z) = -\log_2\left(\max_{x,z} P(X = x | Z = z)\right) \quad (8)$$

where Z represents any side information the adversary possesses that is correlated with X .

In the context of this paper, a concrete and critical instance of such side information is the boot time of a virtual machine. An attacker who can observe or estimate when a VM was started thereby gains knowledge about the state of the entropy pool at that moment, when the pool has had less time to accumulate entropy and the set of plausible RNG states is correspondingly smaller. Formally, $Z=\text{boot time}$ reduces $H_{\infty}(X|Z)$ well below the unconditional $H_{\infty}(X)$, even if the source appears adequate under normal operating conditions.

This is the formal basis for why the boot-time entropy problem identified in the BSI 2016 study constitutes a genuine cryptographic vulnerability rather than a mere statistical artifact. The implications for virtual machine environments and the Linux entropy pool under VMM operation are analyzed in detail in Chapter 4.

2.1.3 IID and Non-IID Sources

The entropy estimation methodology used in this paper, NIST SP 800-90B which is described in more detail in 2.3.3, draws a fundamental distinction between two classes of random sources: IID and Non-IID. This classification determines which analysis track is applied and which estimators are used. Misclassifying a source leads to systematically wrong entropy estimates, and therefore to wrong security conclusions.

A source is IID (Independent and Identically Distributed) if it satisfies two properties simultaneously. The first is independence, meaning that each output is entirely unaffected by those preceding it, so that knowing the last thousand values reveals nothing about the next and the source can be said to have no memory. The second is identical distribution, meaning that every output is drawn from the same probability distribution, so that the bias of the source does not shift over time, across operating conditions, or across samples. A fair coin is the canonical IID source because each flip is independent of the last, and the fifty-fifty distribution never changes.

A source is Non-IID if at least one of these properties is violated. Outputs may be correlated with each other, the distribution may change over time, or both. In practice, most hardware entropy sources are Non-IID for physical reasons that are difficult to eliminate entirely.

Why hardware RNGs are typically Non-IID becomes clear on considering the physical processes involved. In a ring oscillator the current oscillation frequency is slightly influenced by recent behavior, because heat dissipates gradually rather than instantaneously, which gives the circuit what amounts to a thermal memory and leaves consecutive samples not fully independent but carrying a residual correlation between them. If the CPU is running a periodic workload, the interference pattern becomes periodic too, introducing structure into consecutive samples. Most critically for this paper, during the boot phase, oscillators have not yet thermally stabilized so the distribution is actively shifting in the first seconds

of operation, violating the identically distributed requirement directly.

RDRAND, despite producing output that passes standard statistical tests and appears uniform, is built on a physical noise source subject to exactly these effects. The underlying metastable latch operates in a thermal environment that changes over time and under varying CPU load. Although the conditioning and DRBG layers mask this at the output level, the source itself remains Non-IID, so that treating it as IID when estimating entropy would overestimate how much genuine unpredictability the physical process actually contributes.

This classification directly matters for the methodology because NIST SP 800-90B defines two entirely separate analysis tracks. The IID track applies a battery of statistical tests to substantiate the IID claim before a single estimator suffices. The Non-IID track applies ten different estimators simultaneously, each probing a different statistical property of the source, because independence cannot be assumed [11]. The final reported entropy is the minimum across all ten, which is the most conservative result available and reflects the fact that any one of the ten approaches might detect structure the others miss.

Classifying RDRAND as IID and applying the simpler track would be methodologically unsound, since it would skip precisely those tests designed to catch the correlations and distribution shifts that hardware sources exhibit. All RDRAND and RDSEED measurements in this paper are therefore evaluated on the Non-IID track. This is the more conservative choice, and the correct one for any hardware entropy source where the independence of consecutive samples cannot be formally guaranteed.

2.2 Random Number Generator Architecture

2.2.1 The Noise Source, Conditioning and DRBG Pipeline

A cryptographic entropy source is never a single component. It is always a pipeline of three distinct stages, each solving a problem the previous stage cannot. Understanding why each stage exists is essential for understanding what RDRAND actually delivers and why it outputs DRBG-generated pseudorandom bits rather than raw physical entropy.

Noise Source. The first stage is a physical process that generates genuine unpredictability. Thermally induced electron movement, oscillator timing jitter, and radioactive decay are all processes governed by quantum mechanics, thus they are not merely difficult to predict but are in principle non-deterministic. No amount of knowledge about the system's prior state can predict their next output with certainty.

However, raw physical noise signals are not directly usable for cryptography. They are analog rather than digital, slow to produce output, and almost always biased. Thermal noise follows a Gaussian distribution rather than a uniform one. A ring oscillator has slight frequency preferences depending on temperature and load. The result is that raw bits from a noise source carry significantly less than 1.0 bit of min-entropy per

bit of output, in other words they contain genuine randomness, but it is diluted by predictable structure. The noise source is therefore the origin of all entropy in the system, but its output requires further processing before it can be used.

Conditioning. The second stage transforms the raw, biased bits from the noise source into bits that are statistically uniform. This process is called entropy extraction, which is a cryptographic function (typically AES-CBC-MAC or a hash function) that takes a large number of raw input bits and compresses them into a smaller number of output bits. The key property of these functions is that their output appears uniformly distributed whenever the input contains *sufficient* entropy, even if the input itself was not uniform.

What counts as sufficient is precisely quantifiable. If the target is a 128-bit output with 1.0 bit of min-entropy per bit, then the input must contain at least 128 bits of genuine min-entropy. If the noise source delivers 0.6 bits of min-entropy per raw bit, as assessed for the Intel DRNG on Broadwell EP silicon in entropy certificate E65 [12], then at least $\lceil 128/0.6 \rceil = 214$ raw bits must be collected before one conditioning step can produce 128 fully entropic output bits.

The important conceptual point here is that conditioning does not create entropy but rather it repackages it. Before conditioning, 128 bits of genuine entropy are spread across 214 raw bits. After conditioning, the same 128 bits of genuine entropy are concentrated into 128 output bits, each carrying approximately 1.0 bit of min-entropy. The total entropy in the system has not changed, only its density has. After conditioning, Intel's RDRAND achieves approximately 1.0 bit of min-entropy per bit, up from the 0.6 bits per bit of the raw source.

Output that carries at least 1.0 bit of min-entropy per bit is referred to as fully entropic, meaning that no outcome is more predictable than any other and an adversary gains nothing from knowing the distribution of the source.

DRBG. The third stage addresses a practical bottleneck. Even after conditioning, the pipeline is slow. A metastable latch producing a few megabits per second of raw entropy, conditioned down to its entropic core, cannot keep up with the demand of a server handling thousands of TLS connections per second, where each requires fresh random bytes for nonces, session keys, and initialization vectors, with latencies measured in milliseconds.

The Deterministic Random Bit Generator solves this by decoupling production speed from entropy quality. A DRBG is a cryptographically secure algorithm that takes a secret seed, so in this case the conditioned entropy, and expands it into an arbitrarily long sequence of pseudorandom bits. It is seeded once, or periodically reseeded with fresh conditioned entropy, and then produces output at speeds orders of magnitude faster than the physical noise source. Intel's RDRAND uses AES-CTR DRBG as specified in NIST SP 800-90A [13], [14], and the DRBG is periodically reseeded with fresh output from the conditioning stage.

The security guarantee of a DRBG is computational, in that its output is indistinguishable from true randomness for any algorithm running in polynomial time, provided the seed contained sufficient entropy. Two formal properties follow from this [13]. Backtracking resistance ensures that if the internal state at time t is compromised, past outputs cannot be reconstructed. Prediction resistance ensures that after a reseed with fresh entropy, future outputs cannot be predicted even by an adversary who knew the previous internal state.

The full pipeline, with the tap points for RDSEED and RDRAND, is set out in Fig. 2 on the concrete example of the Intel implementation, which Section 3.3.1 describes in detail.

RDSEED taps the pipeline after conditioning but before the DRBG. It delivers conditioned entropy directly, which makes it slower and subject to failure when insufficient raw bits have accumulated. RDRAND taps the DRBG output, which makes it fast and practically infallible because the DRBG always has output ready. The practical implication is that RDSEED is the appropriate source when seeding another DRBG, and RDRAND is the appropriate source for bulk random number generation. Both are examined in this paper, as their different positions in the pipeline have different implications for what entropy measurement can and cannot reveal about the underlying physical source.

2.2.2 A Taxonomy of TRNG, CSPRNG and DRBG

The previous section described the internal pipeline of a cryptographic entropy source. Before applying this to specific implementations, it is useful to establish a precise taxonomy of the generator types involved because the terms TRNG, CSPRNG, and DRBG are often used loosely, and the distinctions between them matter for understanding what security guarantees each type can and cannot provide.

TRNG (True Random Number Generator). A TRNG is any generator whose output derives directly from a physical process that is genuinely non-deterministic. The defining property is that the randomness is physical, not computational, meaning a TRNG’s output cannot be reproduced even by an observer who has complete knowledge of the system’s prior state, because the underlying phenomenon (thermal noise, oscillator jitter, quantum decay) is not determined by prior state. A TRNG corresponds to the noise source stage of the pipeline described in Section 2.2.1. Its output is genuine but raw, meaning that it is slow, analog, and biased.

CSPRNG (Cryptographically Secure Pseudo-Random Number Generator). A CSPRNG is a deterministic algorithm that produces output which is computationally indistinguishable from true randomness. It is also sometimes referred to as a CRNG. The word “pseudo” is the critical qualifier, since given the same internal state a CSPRNG always produces the same output sequence. It is not random in the physical sense but in the computational sense, meaning no efficient algorithm can distinguish its output from a truly random sequence, except with negligible probability. The formal criterion for a CSPRNG is the next-bit test [15], [16]. It states that a generator passes if no polynomial-time algorithm can

predict the next output bit with probability significantly better than one half, given all previous output bits.

The security of a CSPRNG is therefore entirely dependent on its initial state or more precisely, the seed. A CSPRNG seeded with high-entropy input is cryptographically secure. A CSPRNG seeded with low-entropy input, for example, from a weak or compromised noise source, produces output that passes all statistical tests but is predictable to an adversary who can guess or reconstruct the seed. This is the foundational dependency that makes entropy quality measurement so important. In short the CSPRNG amplifies whatever the seed provides, good or bad.

DRBG (Deterministic Random Bit Generator). A DRBG is the formal, standardized instantiation of a CSPRNG as defined by NIST SP 800-90A. Every DRBG is a CSPRNG, but a DRBG additionally specifies a standardized interface, like *Instantiate*, *Generate*, *Reseed*, and must satisfy formally defined security properties including backtracking resistance and prediction resistance as introduced in the previous section. NIST SP 800-90A defines three approved DRBG constructions [13], namely Hash DRBG, HMAC DRBG, and AES-CTR DRBG. Intel RDRAND uses AES-CTR DRBG internally.

In the context of this paper: RDRAND’s metastable latch is the TRNG, the AES-CBC-MAC conditioner extracts entropy from it, and the AES-CTR DRBG produces the actual output. What the caller receives when executing RDRAND is DRBG output. This layering is necessary and well-engineered, but it has a direct consequence for entropy assessment, that is that you are measuring the shadow of the entropy source, not the source itself.

2.2.3 The XOR Combiner Approach and the XOR Hash Lemma

The previous sections established that a single entropy source, however well-engineered, can be weak, biased, or compromised under certain conditions. A noise source may deliver less entropy than specified under thermal stress or high CPU load. In a virtualized environment, a hypervisor may manipulate the values a guest receives. Relying on a single source creates a single point of failure meaning an adversary who compromises that one source has compromised the entire system.

The combiner approach addresses this directly. Instead of trusting one source, the system draws from several independent sources and combines their outputs. The security goal is resilience, meaning that the combined output should remain cryptographically sound as long as at least one source delivers good entropy and the adversary does not control it, regardless of what happens to the others.

How combining works. The simplest form is bitwise XOR. Given multiple entropy sources $S_1, S_2, S_3, \dots$, their outputs are combined as:

$$\text{Output} = S_1 \oplus S_2 \oplus S_3 \oplus \dots$$

The intuition behind XOR as a combiner is elegant, since if even one of the inputs is completely random and unknown to the adversary, the entire output is completely random and unknown as well, XOR with a truly random value being equivalent to a one-time pad. The adversary cannot recover any structure from the output without knowing that one unknown input.

XOR is the simplest and fastest form of combining, but not the strongest. Because XOR is a linear operation, it can be vulnerable under certain algebraic attack conditions. Hash-based combiners, where a cryptographic hash function is applied to all inputs rather than XOR-ing them directly, provide stronger security guarantees because hash functions are non-linear. Linux uses this approach in practice. Its entropy pool applies BLAKE2s, a fast cryptographic hash function, to all incoming entropy inputs before mixing them into the pool state. The concrete Linux implementation is described in detail in Section 3.2.2, here we focus on the underlying principle.

The XOR Hash Lemma as a formal guarantee. The XOR Hash Lemma formalizes the combiner intuition in terms of min-entropy. Its statement is straightforward:

If two sources X and Y are combined via XOR, and X has at least k bits of min-entropy, then the combined output $X \oplus Y$ also has at least k bits of min-entropy, regardless of how predictable Y is, even if Y is completely known to the adversary.

$$H_{\infty}(X \oplus Y) \geq H_{\infty}(X) \quad [8]$$

This guarantee rests on a precondition that is easily overlooked: X and Y must be independent. The adversary may know Y completely, but must not be able to choose it as a function of X . If an attacker who can observe or influence the good source is allowed to set Y adaptively, the lemma collapses, as the extreme case $Y = X$ illustrates, where the combined output $X \oplus X$ is constantly zero although X itself is fully random.

The security guarantee therefore reduces to two conditions. First, there must exist at least one source that delivers genuine entropy and that the adversary neither controls nor observes, and second the remaining sources must not be choosable in reaction to it. If these conditions hold, the combined output is secure. If it does not, if the adversary controls every source, the lemma provides no guarantee because there is no unknown input left to protect the output.

What the combiner cannot do. It's important to know that the combiner is not a filter. It cannot detect that a source is weak or compromised and exclude it. It simply passes the output through, trusting that at least one input is good. This means the security guarantee is only as strong as the weakest link in a rather different sense, the weak point being not the weakest source but the question of whether any source remains outside the adversary's reach.

This is directly relevant for virtualized environments. In a conventional cloud VM, a hypervisor has architectural access to most entropy sources available to the guest meaning

it can distort interrupt timing, manipulate emulated device events, and, as demonstrated in the BSI 2016 study, intercept RDRAND instructions via VM-exits [3]. If the hypervisor controls all of these simultaneously, no source remains that is both good and unknown to it. The combiner's guarantee collapses, because there is no X with high min-entropy that the adversary does not know.

In an Intel TDX Trust Domain, this changes structurally. The hypervisor is excluded from the Trusted Computing Base, and the instruction set a Trust Domain executes is defined by the TDX Module rather than by the host. RDRAND and RDSEED are absent from the instructions that specification restricts [5, Sec. 11.7], the published module source holds both exiting controls at zero and outside the mask a host may modify [6], and Intel's own guest kernel security specification states that the two instructions lie outside host and VMM control [17]. RDRAND therefore becomes a source the hypervisor cannot intercept or manipulate. It is the one input to the combiner that remains outside adversarial control. The XOR Hash Lemma applies again, so that as long as RDRAND delivers genuine entropy the combined pool output is cryptographically secure, regardless of what the hypervisor does to the other sources.

This is the formal reason why the entropy quality of RDRAND under confidential computing is the central question of this paper. In a confidential cloud environment, RDRAND is not merely one of several sources, it is the only source the combiner can rely on. Whether an entropy assessment of the delivered output can establish that it fulfills that role is precisely the question the empirical part of this work addresses. That assessment covers bare metal on both vendor platforms and a SEV-SNP guest on the AMD platform, and it does not cover a Trust Domain, for the reasons Chapter 6 sets out. Sections 7.1 and 7.7 report what it found, and Section 7.2 establishes how much such a finding can carry.

2.3 Relevant Standards

Three standards frame the technical and regulatory context of this paper, each addressing a different layer of the same problem. NIST SP 800-90A defines how a DRBG must be constructed. It is the implementation framework that specifies which algorithms are approved and what security properties they must satisfy. NIST SP 800-90B defines how to measure whether the entropy source feeding that DRBG is actually good enough. Because the empirical analysis in this paper rests on exactly this kind of measurement, SP 800-90B serves as the operative measurement tool throughout the following chapters. BSI AIS 20/31 defines how a complete random number generation system is evaluated within a formal certification scheme. Together the three standards describe the full chain from physical entropy source through algorithmic construction to certified system, and understanding all three is necessary to correctly interpret both the measurements and their implications. Before turning to the standards themselves, Section 2.3.1 establishes the regulatory framework within which they operate, because concepts such as FIPS validation, the CMVP, and the security levels of FIPS 140-3

recur throughout this paper and determine what the individual standards are able to guarantee in practice.

2.3.1 Regulatory Framework

Federal Information Processing Standards, or FIPS, are mandatory requirements for IT systems used by US federal agencies, developed by NIST and made binding by the US Department of Commerce [18]. In the context of cryptography, two fundamentally different relationships to a FIPS standard are possible, and the distinction between them is significant for the analysis in this paper. A system is FIPS-compliant if its developer has implemented an algorithm or construction according to a FIPS standard, which is a self-assessed claim that carries no external verification. A system is FIPS-validated if an accredited independent laboratory has tested it against the standard and NIST has formally confirmed the result, so that an independent party has verified the claim rather than merely accepted it. Compliance says the developer followed the rules, while validation says someone checked.

The program through which FIPS 140 validations are issued is called the Cryptographic Module Validation Program, or CMVP, operated jointly by NIST and the Canadian Centre for Cyber Security [19]. A manufacturer submits a cryptographic module to an accredited laboratory, which tests it against FIPS 140-3, and if the module meets all requirements it is listed in the publicly accessible CMVP registry that documents not only which modules have been validated but also their associated entropy assessments under NIST SP 800-90B. Since 2022 the CMVP additionally operates the Entropy Source Validation program, or ESV, through which an entropy source is validated against SP 800-90B independently of any complete module [20], allowing a single validated source to be referenced by multiple module validations. For this paper, the CMVP serves two concrete purposes. It documents what a vendor certificate actually assesses, namely the raw output of a noise source before conditioning, which is a different object from the conditioned DRBG output an instruction such as RDRAND returns and which this paper measures, so that the certified figures are context for the assessment rather than reference values it could be compared against. Beyond that, the scope and limits of the validation that RDRAND has actually obtained form one of the central trust questions examined in Chapter 5.

FIPS 140-3 organizes validated modules into four security levels that describe how well the physical device resists attack [18]. Level 1 sets minimal requirements and can be met by software implementations with no physical protection. Level 2 adds tamper-evident mechanisms such as seals that reveal whether a device has been opened. Level 3 requires active tamper resistance such that the device automatically destroys all sensitive data if an attempt is made to open or manipulate it physically. Level 4 provides the highest degree of physical security, requiring that penetration attempts are detected and answered with immediate zeroization of sensitive data with very high probability, and that the module withstands environmental attacks such as manipulation of voltage or temperature beyond its normal operating range. In practice, the crypto-

graphic HSMs deployed in high-security environments are typically Level 3 devices, and it is this level that defines the trust guarantees a hardware security module is expected to provide.

A Hardware Security Module is a dedicated physical device designed to perform cryptographic operations and protect sensitive key material, and what distinguishes it from a software solution or a CPU-integrated solution like RDRAND is the combination of three mutually reinforcing properties. Physical isolation means the HSM is a self-contained device with its own processor, memory, and entropy source that shares no resources with the host system and operates in a protected perimeter entirely separate from the general-purpose computing environment. Tamper resistance at Level 3 means that any physical attempt to open or probe the device triggers automatic erasure of all key material and entropy state, so that secrets cannot be extracted by physical means without being destroyed first. Independent certification means that the complete pipeline encompassing the physical entropy source, the conditioning stage, the DRBG, and the key storage has been examined and verified by an accredited laboratory, with the results documented publicly in the CMVP registry so that any relying party can inspect the evidence independently.

RDRAND occupies a structurally different position that is neither a software solution nor a dedicated external hardware device. The entropy source is a physical hardware circuit embedded in the CPU die, so genuine physical randomness is involved, and since 2023 Intel holds an ESV certificate for its DRNG entropy source [21], confirming that the statistical behavior and design documentation of the source satisfy the requirements of SP 800-90B. What this certificate covers, however, is deliberately narrow. ESV validation rests on statistical testing of raw noise data and on design documentation, both of which are supplied by the manufacturer, and it involves no physical inspection of the silicon, because the metastable latch and conditioning circuitry are entirely enclosed within the CPU package and inaccessible to external examination. Beyond this, RDRAND has no physical protection perimeter of its own and shares the CPU package with all other components of the host system, which rules out FIPS 140-3 Level 2 or higher regardless of the quality of its internal design. An accredited laboratory can therefore confirm that RDRAND behaves like a sound entropy source, but it cannot independently verify that the implementation inside the silicon is what Intel documents it to be, and trusting RDRAND consequently still reduces in large part to trusting Intel as the sole authority on its own implementation.

HSMs themselves offer no straightforward escape from this situation in virtualized environments, because accessing a physical HSM from within a virtual machine requires either mediation by the hypervisor or trust in a cloud provider's API path, both of which re-introduce exactly the kind of external trust dependency the HSM architecture was designed to eliminate. This structural gap, developed in detail in Chapter 1, motivates the empirical comparison undertaken in this paper, which asks not whether RDRAND can obtain a physical certification that its architecture rules out, but what

an entropy assessment of its output can establish at all and whether Intel TDX closes the hypervisor-based interception vector architecturally.

2.3.2 NIST SP 800-90A and DRBG Construction

Before SP 800-90A existed, there was no unified picture of what a cryptographically secure random number generator must look like. Different systems implemented DRBGs in different ways, without common security requirements or standardized interfaces, which made it difficult to compare implementations or reason about their security properties. SP 800-90A addressed this by defining which DRBG constructions are approved for cryptographic use, what formal security properties they must satisfy, and what interface they must expose. In the FIPS context, compliance with SP 800-90A is not optional, so any software or hardware deployed in US government systems must use one of the constructions the standard approves.

The standard defines three approved constructions. Hash DRBG is based on a cryptographic hash function such as SHA-256 or SHA-512, with the internal state transformed through repeated hashing. It is straightforward to implement and well-analyzed. HMAC DRBG is structurally similar but uses a Hash-based Message Authentication Code, and its security proof is more robust in the sense that it holds under weaker assumptions about the underlying hash function, which makes it the preferred choice in many software implementations. AES-CTR DRBG is based on AES operating in counter mode, where a secret key and a counter are encrypted by AES to produce the output. It is the fastest of the three constructions because AES executes at hardware speed on modern processors through dedicated instructions such as AES-NI. Intel RDRAND uses AES-CTR DRBG internally. All three constructions are cryptographically sound, so the choice between them is primarily a question of performance and implementation context, not of relative security strength.

Regardless of which construction is used, every DRBG defined by SP 800-90A must expose a standardized interface consisting of three operations. Instantiate initializes the DRBG with a seed drawn from an entropy source, establishing the initial internal state. Without sufficient entropy at this stage, the DRBG is insecure from the start, the internal state is predictable, and all subsequent output follows deterministically from it. Generate produces pseudorandom bits from the current internal state, which is updated deterministically with each call. The Generate operation can be called repeatedly until a maximum reseed interval is reached, which for AES-CTR DRBG is set at 2^{48} calls by the standard. In practice, RDRAND reseeds far more frequently than this limit requires, because the conditioning stage continuously supplies fresh entropy. Reseed renews the internal state with fresh entropy, restoring the security guarantees that would otherwise weaken after many Generate calls. RDRAND performs this step automatically whenever the conditioning stage has accumulated sufficient new entropy from the physical noise source.

Two formal security properties follow from the construction requirements. Backtracking resistance ensures that an adver-

sary who compromises the internal state at time t cannot reconstruct any outputs generated before that point, because the DRBG updates its state irreversibly at every Generate call. Prediction resistance ensures that after a Reseed with genuine fresh entropy, an adversary who knew the previous internal state cannot predict future outputs, because the new state depends on entropy the adversary does not know.

SP 800-90A functions as an implementation framework rather than a measurement tool, specifying how a DRBG must be built and what properties it must satisfy while explicitly leaving open how the entropy feeding that DRBG is generated or validated. Because the standard assumes that sufficient entropy is available at Instantiate and Reseed time without verifying this assumption, it deliberately separates the algorithmic layer from the question of entropy source quality, a concern addressed instead by SP 800-90B. The AES-CTR DRBG that Intel implements for RDRAND is constructed according to an approved standard, which establishes that the algorithmic layer of RDRAND is cryptographically legitimate, yet whether the physical entropy source seeding that DRBG actually delivers sufficient entropy remains a question SP 800-90A cannot answer. Addressing that question empirically is the task this paper takes on.

The current version of the standard is Rev. 1, published in 2015, and it is the version that RDRAND implements and that all current FIPS validations reference. NIST is preparing a second revision which will introduce a new DRBG construction based on extendable output functions, specifically the SHAKE algorithms from the SHA-3 standard. Unlike classical hash functions that produce a fixed-length output, extendable output functions can produce output of arbitrary length by directly stretching the internal state, which avoids the need for repeated hashing. Rev. 2 has not yet been finalized and its adoption will involve a transition period of at least one year after publication. Crucially, it does not deprecate or replace any of the three existing constructions but rather adds a fourth option. AES-CTR DRBG remains fully approved under both the current and forthcoming standard, and Rev. 1 remains the reference for all measurements and comparisons in this paper.

2.3.3 NIST SP 800-90B and Entropy Estimation

SP 800-90A deliberately assumes that sufficient entropy is available when a DRBG is instantiated or reseeded, and SP 800-90B is the companion standard that makes this assumption testable. Its subject is not the DRBG but the entropy source feeding it, and its central question is how much unpredictability a physical source actually delivers per bit of output. The measure the standard uses for this is min-entropy to capture the success probability of an adversary who always guesses the single most probable value.

The standard defines two assessment tracks that differ in what may be assumed about the source. If the output samples can be shown to be independent and identically distributed, a property introduced in Section 2.1.3, the source qualifies for the IID track, in which a battery of statistical tests must first substantiate the IID claim before a single estimator based on the frequency of the most common value suffices to determine

the entropy. Physical sources rarely justify this assumption, because effects such as temperature drift, aging, and coupling between consecutive samples introduce dependencies, so the non-IID track is the realistic path for most hardware entropy sources and the one relevant throughout this paper. In this track no structural assumptions are granted, and the source must instead survive a full set of ten estimators, each designed to detect a different kind of statistical weakness.

The ten non-IID estimators fall into three groups that approach predictability from different directions. The first group measures simple distributional defects, with the Most Common Value estimator deriving entropy directly from the frequency of the most likely symbol, while the Collision, Markov, and Compression estimators, applied to binary data, examine how quickly repeated values occur, how strongly each bit depends on its predecessors, and how far the data can be compressed. The second group, consisting of the t-Tuple and Longest Repeated Substring estimators, searches for recurring patterns of increasing length, since frequently repeating tuples indicate structure an adversary could exploit. The third group consists of four prediction-based estimators, MultiMCW, Lag, MultiMMC, and LZ78Y, which train simple predictive models on a prefix of the data and derive the entropy estimate from how often these models guess the next sample correctly, the intuition being that any output a machine-learning-like predictor can anticipate is output an adversary can anticipate as well.

The final entropy assessment is the minimum over all applicable estimators, and this construction is the reason the result must always be read as a lower bound. Each estimator targets a specific class of defects, and by taking the minimum the standard credits the source with no more entropy than its weakest observable property supports, effectively assuming an adversary who exploits whatever structure is most detectable. The methodology therefore errs systematically toward underestimation, which is the appropriate direction of error for a security standard, because a source whose entropy is understated leads to overly conservative designs while a source whose entropy is overstated leads to broken keys. For the interpretation of the measurements in this paper this asymmetry matters in both directions, since a low estimate does not prove that the underlying physics is weak, whereas a high estimate that holds across all ten estimators constitutes strong evidence that no statistically detectable weakness is present in the observed output.

Beyond the one-time entropy assessment, the standard mandates health tests that run continuously during operation, because a source that was healthy at evaluation time can fail in the field through physical degradation, environmental stress, or deliberate manipulation. Two continuous tests are required, with the Repetition Count Test detecting a source that becomes stuck and emits the same value repeatedly, and the Adaptive Proportion Test detecting a sudden loss of variability in which one value begins to dominate the output. These tests are deliberately coarse, designed not to measure entropy precisely but to catch catastrophic failure quickly, and the standard additionally requires start-up tests before the first

output is used and permits vendor-defined tests tailored to the specific source. Intel's DRNG implements such health tests in hardware on the raw output of its noise source, a design aspect that becomes relevant in Chapter 3.

SP 800-90B serves as the primary measurement tool of this paper for two reasons that follow directly from its design. First, its estimators operate purely on output data and require no access to the internal construction of the source, which matches exactly the epistemic position of a tenant inside a virtual machine who can execute RDRAND but cannot inspect the silicon, and it is this black-box applicability that makes an independent empirical assessment possible at all. Second, the entropy certificates published through the CMVP and ESV programs rest on this same methodology, so applying it to RDRAND means applying the procedure that vendors and validators already regard as authoritative for this class of source. What separates the two applications is the object each is applied to. The standard is designed for the raw output of a noise source, whereas RDRAND emits conditioned output that has passed through cryptographic post-processing, so estimates obtained on RDRAND output characterize the delivered output stream rather than the physical source behind it. The certified figures are therefore not values this paper measures against but a point of contrast, and what an assessment of conditioned output can establish at all is examined in Chapter 6.

2.3.4 BSI AIS 20/31

While the SP 800-90 series defines the American regulatory perspective, AIS 20/31 is its German counterpart, developed by the Federal Office for Information Security, the BSI, as the binding evaluation methodology for random number generators within the German Common Criteria certification scheme. The two-part designation reflects a division of labor along the same line that separates 90A from 90B, in that AIS 20 covers deterministic random number generators, whose output is computed algorithmically from a seed in the manner of the DRBGs described in Section 2.3.2, while AIS 31 covers true random number generators, whose output originates in a dedicated physical noise source of the kind SP 800-90B is designed to measure. Both documents operate within a different certification world than FIPS, since Common Criteria evaluations assess complete products against protection profiles rather than validating cryptographic modules in isolation, yet they address the same underlying question of when the output of a generator may be trusted for cryptographic use.

The most distinctive structural choice of AIS 20/31 is that it defines no approved designs at all. Where SP 800-90A names three concrete constructions and requires implementers to choose among them, AIS 20/31 is technology-neutral and instead defines functionality classes, each consisting of security requirements that a generator of any internal design must demonstrably fulfil, with the evidence provided jointly by the applicant and an accredited evaluation laboratory. The deterministic classes DRG.2, DRG.3, and DRG.4 form an ascending hierarchy [22] in which DRG.2 guarantees backward and forward secrecy, DRG.3 adds enhanced backward secrecy

so that a compromised internal state does not reveal earlier outputs, and DRG.4 adds enhanced forward secrecy through mandatory reseeding from a true randomness source. The physical classes PTG.2 and PTG.3 apply to generators built on physical noise, with PTG.3 as the strongest class in the entire scheme, requiring a strong physical noise source combined with cryptographic post-processing. The class NTG.1 covers non-physical true generators that harvest entropy from unpredictable system events, the category to which the Linux kernel generator examined in this paper belongs.

For the physical classes, AIS 31 imposes a requirement that has no counterpart in the NIST methodology and that marks the deepest philosophical difference between the two frameworks. The developer must provide a stochastic model of the noise source, meaning a formal mathematical description of the physical process from which the randomness originates, from which a justified lower bound on the entropy of the raw output can be derived and against which the online tests monitoring the source during operation must be tailored. Where SP 800-90B estimates entropy from the outside by applying statistical estimators to output data without requiring any understanding of the underlying physics, AIS 31 demands an explanation from the inside, accepting a physical generator only if its randomness can be traced to a modeled and quantified physical phenomenon. The two approaches are complementary rather than competing, since a stochastic model without empirical confirmation rests on assumptions while measurements without a model cannot distinguish genuine physical entropy from the output of a well-seeded deterministic generator, and it is precisely this complementarity that the certification communities on both sides have begun to formalize.

The framework has recently been modernized in a way that matters for the currency of this paper. In September 2024 the BSI published version 3.0 of the mathematical-technical reference underlying both AIS 20 and AIS 31 [22], unifying the previously separate methodologies in a single document, extending the class system by DRT.1 for trees of deterministic generators that seed one another, and explicitly analyzing how the NIST-approved DRBG constructions relate to the requirements of the deterministic classes. In parallel, BSI and NIST are actively harmonizing their frameworks, and the report NIST IR 8446, finalised in January 2026, serves as the bridge document that maps the AIS 20/31 functionality classes onto the constructions of the SP 800-90 series and explains where the requirements coincide and where they diverge [23]. The practical consequence of this convergence is that evidence gathered under one framework has become legible under the other, so that SP 800-90B entropy estimates, while not replacing an AIS 31 evaluation, are recognized within it as a meaningful component of the empirical evidence.

For this paper, AIS 20/31 defines the evaluation context in which the results are intended to be read, and its perspective sharpens rather than merely duplicates the NIST view. Architecturally, RDRAND follows the pattern that AIS 31 captures in its strongest class, a physical noise source whose output is cryptographically post-processed, yet an actual PTG.3 evalu-

ation of RDRAND is impossible for the same reason identified in Section 2.3.1, because the stochastic model it would require presupposes a level of insight into the physical implementation that Intel does not publicly provide and that no external party can independently obtain. The measurements presented in this paper therefore occupy a precisely definable position within the AIS framework, in that they deliver the empirical, output-based half of the evidence an AIS 31 evaluation would demand while making the absence of the model-based half explicit, and presenting the results in these terms ensures that the methodology remains compatible with the evaluation scheme of the intended audience.

3 ARCHITECTURE OF NOISE SOURCES

This chapter describes the architecture of the entropy sources relevant to this study. Section 3.1 introduces the general noise source model established by the BSI 2016 study [3], which serves as the analytical framework throughout this paper. Sections 3.2 through 3.4 then apply this model to the specific implementations under investigation: the current Linux random number generator, and the Intel and AMD RDRAND/RDSEED implementations.

3.1 General Noise Source Architecture

Noise sources exist in many different forms, among them dedicated hardware circuits on smart cards or HSMs, CPU instructions such as RDRAND, and software routines that observe the timing of hardware events, yet regardless of their implementation they all follow a common architectural pipeline. The BSI 2016 study [3] formalized this pipeline in a model that is consistent with the noise source architecture defined in NIST SP 800-90B [11] and BSI AIS 20/31 [22]. Section 2.2.1 introduced the same pipeline in its three-stage conceptual form, noise source, conditioning, and DRBG, which was sufficient to explain what RDRAND delivers. The model used here refines that account into the six stages an evaluator must treat separately, since recording, digitization, and health testing are distinct points of attack and of failure even though they add no entropy of their own. It is this finer decomposition that the analysis of VMM interference in Chapter 4 operates on. This model decomposes any noise source into the stages set out below, and Fig. 1 shows them in the order in which a design passes through them.

Unpredictable Phenomenon. At the origin of every noise source is a physical or environmental phenomenon that exhibits behavior that is not fully predictable to an observer. This may be a fundamentally stochastic process such as thermal noise, shot noise, or the resolution time of a metastable latch, or else an environmental process whose precise timing cannot be predicted with sufficient accuracy, such as the exact arrival time of hardware interrupts or the seek latency of a spinning disk. The entropy of the entire pipeline is ultimately bounded by the unpredictability of this phenomenon, since no subsequent processing stage can add entropy but only preserve or compress it.

Recording. A recording mechanism captures the output of the unpredictable phenomenon in a form suitable for further processing. For hardware noise sources, this may be a sampling circuit that reads the state of a ring oscillator at fixed intervals. For software noise sources, recording typically consists of obtaining a high-resolution timestamp at the moment a hardware event occurs. The recording stage bridges the physical world and the digital domain, determining both what aspect of the phenomenon is captured and at what resolution.

Digitization. The recorded data is converted into a digital bitstream. In hardware noise sources, this may be as simple as reading the binary output of a comparator. In software noise sources, digitization commonly involves interpreting a timestamp or event descriptor as a byte sequence and injecting it into an entropy pool. After digitization, the data is in a form that can be processed by standard computational operations.

Health Testing. A health test monitors the noise source output for signs of malfunction, such as a sequence of identical values that would indicate the physical source has failed or become stuck. Health tests are designed to detect catastrophic failures, not to measure the entropy rate of a functioning source. NIST SP 800-90B mandates two continuous health tests for entropy sources: the Repetition Count Test, which detects repeated identical outputs, and the Adaptive Proportion Test, which detects an output value occurring disproportionately often within a sliding window [11]. AIS 20/31 imposes similar requirements [22].

Conditioning. A conditioner takes the raw digitized output, which may be biased or have less than one bit of entropy per bit of output, and compresses it into a shorter sequence with higher entropy density. The entropy accounting of this step was developed in Section 2.2.1. Common conditioning mechanisms include cryptographic hash functions (such as AES-CBC-MAC in Intel RDRAND, or BLAKE2s in the current Linux kernel), linear feedback shift registers (LFSRs), and Von Neumann debiasing. Conditioning does not create entropy but redistributes existing entropy across fewer output

bits, ideally approaching one bit of entropy per bit of output. A conditioner with a compression ratio of 2:1, for example, takes 512 bits of raw input and produces 256 bits of conditioned output, roughly doubling the entropy per bit.

Deterministic Random Bit Generator (DRBG). The final stage is a cryptographically secure deterministic generator that expands the conditioned entropy into a high-rate output stream suitable for cryptographic consumption, with the security properties set out in Section 2.2.2. Common constructions include AES-CTR DRBG (used in Intel RDRAND and specified in NIST SP 800-90A [13]) and ChaCha20 (used in the current Linux CRNG). The DRBG is periodically reseeded with fresh entropy from the preceding stages to maintain its security properties. Between reseeding events the DRBG output is entirely deterministic, so that its security rests on the assumption that the internal state is unknown to any adversary.

It is important to note that the conditioner and DRBG are entropy-neutral, in that they transform and expand existing entropy without generating any, so that all entropy originates exclusively in the unpredictable phenomenon. This has a direct consequence for entropy measurement, because statistical tests applied to the output of a well-designed DRBG will always indicate near-perfect randomness regardless of whether the underlying noise source is functioning correctly. A DRBG with a compromised or low-entropy seed produces output that is statistically indistinguishable from genuine randomness but is nonetheless predictable to an adversary who knows or can guess the seed. This is the fundamental reason why entropy assessment must wherever possible target the raw noise source output rather than the post-processed DRBG output, and equally the reason why the opacity of RDRAND’s internal pipeline, which exposes only DRBG output, constitutes a methodological limitation that Section 6 addresses explicitly.

For the analysis in subsequent chapters, the distinction between hardware-implemented and software-implemented noise sources is critical. Where a noise source is implemented entirely in hardware, as it is for Intel and AMD RDRAND, all

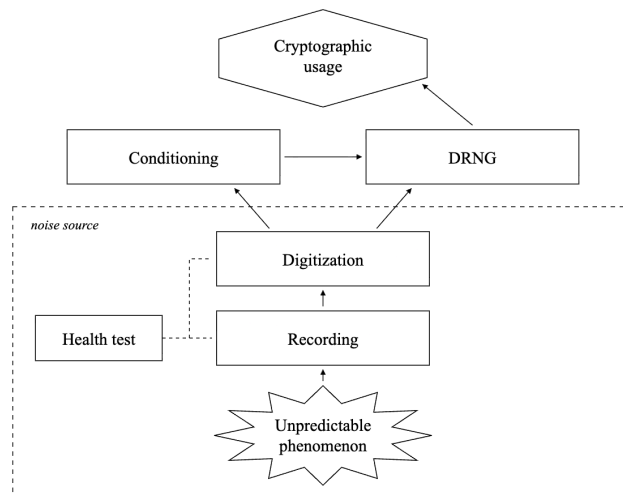


Fig. 1. Architecture of Noise Source

pipeline stages from the unpredictable phenomenon through the DRBG execute within the CPU silicon, so that a virtual machine monitor cannot interfere with the internal operation of these stages and can only intercept the CPU instruction through which software retrieves the final output. Where a noise source is implemented partially or entirely in software, as it is for the Linux kernel’s interrupt-based entropy collection, the recording, digitization, and health testing stages execute as kernel code while the unpredictable phenomenon originates in hardware events. The VMM can then interfere at the boundary between hardware and software, since it is able to alter the timing characteristics of events, to intercept the high-resolution timestamp instruction, or to manipulate the hardware events themselves through device emulation. This architectural distinction is the foundation for understanding why VMMs affect different noise sources in fundamentally different ways, as analyzed in Chapter 4.

3.2 The Current Linux Random Number Generator

3.2.1 */dev/random and /dev/urandom in Current Kernels*

The Linux kernel exposes its random number generator to user space through three primary interfaces: the device files `/dev/random` and `/dev/urandom`, and the `getrandom()` system call. While the device file names have remained unchanged since their introduction in 1994, the underlying architecture and behavior of these interfaces have been fundamentally redesigned in recent kernel versions. The analysis in this section is based on the current BSI documentation of the Linux RNG [24], which covers Linux kernel 6.18.

Architectural overview. The Linux RNG operates around a single central entropy pool, the input pool, which is implemented as a BLAKE2s message digest instance [24]. All entropy collected from the various noise sources is hashed into this pool. A base ChaCha20-based DRNG (“base CRNG”) is periodically seeded from the input pool. To avoid contention on multi-core systems, a set of secondary ChaCha20 DRNGs, one per online CPU, are seeded from the base CRNG, and these secondary instances ultimately provide the random bits delivered to callers. All user-space interfaces draw their output from the same underlying ChaCha20 DRNGs, so that the differences between them lie not in the cryptographic strength of the output but in their timing and blocking behavior with respect to the initial seeding state.

`/dev/urandom`. The `/dev/urandom` device file provides unrestricted access to random bytes. Reads from `/dev/urandom` never block, since data is delivered regardless of the current entropy state of the system. Internally, `/dev/urandom` is identical to the in-kernel function `get_random_bytes()`, which is the recommended interface for kernel-internal callers. Under the previous Linux RNG architecture analyzed in the BSI 2016 study, `/dev/urandom` was often regarded as a “less secure” alternative to `/dev/random` because it could produce output before sufficient entropy had been accumulated. Under the current architecture the distinction disappears once the base CRNG is fully seeded, after which `/dev/urandom` and `/dev/random` produce cryptographically equivalent output.

`/dev/random`. The `/dev/random` device file behaves identically to `/dev/urandom` in steady-state operation. It blocks only once during the lifetime of the system, at early boot, until the base CRNG has accumulated at least 256 bits of initial entropy [24], after which it operates non-blocking for the remainder of the system’s uptime, drawing from the same secondary ChaCha20 DRNGs as `/dev/urandom`. This is a fundamental departure from the legacy behavior, in which `/dev/random` tracked an ongoing entropy estimate and blocked whenever the estimated entropy fell below a configurable threshold. The historical recommendation to “use `/dev/random` for cryptographic key generation and `/dev/urandom` for everything else” is no longer meaningful under the current architecture.

`getrandom()`. The `getrandom()` system call, introduced in kernel 3.17 and now the recommended interface for user-space cryptographic applications, provides the same underlying functionality as the device files but with a more flexible API. When called with a flags field of zero, `getrandom()` behaves like `/dev/random` in that it blocks at early boot until 256 bits of initial entropy have been collected and operates non-blocking thereafter. When called with the `GRND_INSECURE` flag, it behaves like `/dev/urandom` and never blocks. A third flag, `GRND_RANDOM`, exists but is unused in kernel 6.18. The primary practical advantage of `getrandom()` over the device files is that it requires no file descriptor and cannot fail in environments where `/dev` is unavailable, such as restricted containers or early init contexts.

Comparison to the BSI 2016 architecture. The system analyzed in the BSI 2016 study used a fundamentally different design, built on three separate entropy pools named `input_pool`, `blocking_pool`, and `nonblocking_pool`, implemented as LFSRs with SHA-1 as the extraction function, together with an ongoing entropy estimate that gated `/dev/random` output and a sharper qualitative distinction between blocking and non-blocking interfaces. The current architecture replaces this with a single BLAKE2s-based input pool, a hierarchy of ChaCha20 DRNGs as output stages, and a single-shot initial seeding threshold rather than continuous entropy accounting [3], [25]. This is not an incremental refinement but a complete redesign, motivated by the recognition that once a cryptographically secure DRBG is properly seeded, drawing additional output from it does not reduce its security, which made the legacy entropy depletion model that blocked `/dev/random` unnecessarily restrictive. The implications of this architectural change for the BSI 2016 findings on virtualized environments are discussed in Chapter 4.

The next section describes how the Linux RNG combines entropy from multiple noise sources into the input pool, and what cryptographic guarantees this combiner provides.

3.2.2 *How Linux Mixes Entropy Sources*

The Linux random number generator does not rely on a single noise source. Instead, it draws entropy from multiple independent sources in parallel and mixes them into the input pool through a cryptographic combiner. This design addresses the central question of how the system can remain secure when one or more entropy sources are compromised,

biased, or under adversarial control, and the answer lies in the cryptographic properties of the combiner, which guarantees that the combined output is at least as unpredictable as the strongest individual source. This property is foundational for the trust analysis in subsequent chapters and forms a structural counterargument to concerns about the trustworthiness of any single hardware entropy source, including RDRAND.

The current Linux RNG accepts entropy from a heterogeneous set of independent sources, comprising both software-derived inputs (such as interrupt timing and block device events) and hardware-derived inputs (such as RDRAND and other hardware random number generators). A complete enumeration of all entropy sources contributing to the input pool is given in the following Section 3.2.3. For the purposes of the present discussion, the relevant property is that the set of sources is heterogeneous and that the sources are assumed to be mutually independent.

Mechanism of the combiner. All entropy inputs are absorbed into the input pool through the BLAKE2s hash function introduced in Section 3.2.1. Each new input is incorporated via a BLAKE2s update operation, which incrementally extends the internal hash state with the new data. Mathematically, this is equivalent to concatenating all entropy contributions into a single byte sequence and computing BLAKE2s over the entire concatenation:

$$\text{pool_state} = \text{BLAKE2s}(s_1 \parallel s_2 \parallel s_3 \parallel \dots \parallel s_n) \quad (9)$$

where $s_1, s_2, \dots, s_n$ are the individual entropy contributions from the various sources. The output of the input pool is then used to seed the base CRNG. Because the hash function is sensitive to every bit of its input, any change in any single contribution propagates to the entire pool state, so that no source can be skipped or averaged out by the combiner.

Cryptographic guarantee. The security of this construction rests on the so-called XOR Hash Lemma [26], [8], which establishes that the output of a cryptographic combiner over n independent entropy sources retains at least the min-entropy of the strongest individual source. Formally, for independent sources $X_1, X_2, \dots, X_n$ combined via a suitable cryptographic combiner C :

$$H_\infty(C(X_1, X_2, \dots, X_n)) \geq \max_i H_\infty(X_i) \quad (10)$$

The full theoretical treatment of this result, including the precise conditions under which it holds, is presented in Section 2.2.3. The crucial practical implication is that the combined output is secure as long as **at least one** source provides sufficient entropy, regardless of the state of all other sources. An adversary who fully controls $n-1$ of the n sources, whether by predicting their outputs, by replacing them with values of their choosing, or by learning them through a side channel, still cannot predict the combined output, provided the remaining source is unpredictable and the controlled inputs are chosen without knowledge of it, which preserves the independence assumption established in Section 2.2.3. The combiner takes the **maximum** of the individual entropies, not the average.

Implications for the RDRAND trust question. This property constitutes a structural answer to the criticisms of RDRAND

discussed in Section 1.2. Even if RDRAND were compromised by a backdoor, biased through a manufacturing defect, or fully predictable to a sophisticated adversary, the security of the Linux RNG output would not be undermined, provided that at least one of the other entropy sources contributing to the input pool remains uncompromised. The trust requirement is therefore not “trust RDRAND” but rather “trust that at least one of the entropy sources is honest.”

This guarantee comes, however, with the critical caveat that the sources must be **genuinely independent** and must provide **non-trivial entropy**. In virtualized cloud environments, this assumption becomes problematic. As the BSI 2016 study demonstrated and Chapter 4 analyses in detail, several of the software-derived entropy sources degrade significantly in VMs, since interrupt timing comes under the control of the hypervisor, block device events lose their physical-disk timing characteristics, and HID events are absent entirely. When most non-RDRAND sources are weak or under adversarial control, the security of the combined output depends heavily on RDRAND itself, which partially negates the structural protection that the combiner is designed to provide. This tension between the theoretical guarantee of the combiner and the practical entropy landscape of cloud VMs is one of the central concerns of this paper.

A related operational consideration is how the kernel accounts for the entropy contributed by RDRAND. The Linux kernel maintains an entropy estimate for the input pool, used to determine when the base CRNG has been sufficiently seeded for cryptographic use. By default the kernel credits RDRAND output with full entropy, treating each bit of RDRAND output as contributing one bit of min-entropy to the pool. This behavior is governed by the boot parameter `random.trust_cpu`, which defaults to `y` on most modern distributions [24]. When set to `n`, RDRAND output is still mixed into the pool but is not credited with any entropy, forcing the kernel to rely exclusively on other sources for the initial seeding threshold. This configuration option directly reflects the trust assumption an operator places in RDRAND, since trusting it accelerates initial seeding while distrusting it provides defense-in-depth at the cost of potentially significant boot-time delays, a tradeoff with concrete consequences for cloud workloads that Chapter 4 examines further.

The next section describes the architecture of RDRAND and RDSEED in detail, since their role as one of the most consequential entropy sources in modern Linux warrants a thorough treatment of their internal design.

3.2.3 Entropy Sources Contributing to the Input Pool

The Linux random number generator deliberately draws from a heterogeneous set of independent entropy sources rather than relying on any single source. Each source is integrated into the input pool through a dedicated kernel function that exposes a specific class of system events to the entropy collection mechanism. This section enumerates the entropy sources present in the current Linux kernel (version 6.18) and characterizes the role of each. The complete technical reference for these sources is given in Section 3.5 of [24], whereas the descrip-

tion here focuses on the properties relevant to the analysis in subsequent chapters. A key observation is that the entropy landscape has expanded substantially since the BSI 2016 study, which considered only the three sources of interrupts, disk I/O, and HID events, because several additional sources have since been introduced to address weaknesses identified in later work, most notably the snapshot-related vulnerabilities documented by Ristenpart and Yilek [27].

The sources can be grouped into four functional categories. **Software-derived timing sources** observe the timing of hardware events and extract entropy from the unpredictability of those timings, a category comprising interrupt timing, disk I/O completion timing, and HID event timing. **Identifier sources** contribute device-specific values that personalize the entropy pool without being credited as entropy. **Hardware-derived sources** include outputs from dedicated hardware random number generators, either CPU-internal (RDRAND, RDSEED) or externally exposed via the kernel's hwrng framework. **Lifecycle and auxiliary sources** address specific operational moments such as system boot, virtual machine forking, and scheduler activity, all of which contribute entropy outside the regular hardware event stream.

Interrupt timing (`add_interrupt_randomness`). This is the primary software-derived entropy source on headless server systems, where keyboard and mouse activity is absent. The kernel invokes `add_interrupt_randomness` from every hardware interrupt handler, capturing the high-resolution timestamp of the event together with the interrupt number and register state. To minimize per-interrupt overhead the function writes into a per-CPU data structure named `fast_pool`, and these per-CPU buffers accumulate interrupt data using SipHash before being periodically transferred into the BLAKE2s input pool. The entropy derived from interrupt timing depends critically on the unpredictability of the underlying hardware event arrivals, which is one of the central concerns when analyzing this source in virtualized environments (see Chapter 4).

Disk I/O completion timing (`add_disk_randomness`). This source captures the completion timing of block device operations. The kernel maintains a separate timer state structure per physical disk, treating each disk as an independent entropy source. On systems with traditional rotating disks, this source historically provided substantial entropy due to the mechanical variability of seek times. On modern systems dominated by solid-state storage this contribution is significantly reduced, because SSD access times are far more deterministic than HDD seek times. In server environments, where storage often runs on SSDs or is mediated through virtualized block devices, the practical contribution of this source has diminished accordingly.

HID event timing (`add_input_randomness`). This source captures the timing and values of human interface device events such as keystrokes, mouse movements, and button presses. It is a primary source on interactive systems but typically unavailable on headless servers and cloud VMs, which lack attached input devices. The kernel maintains a

single timer state instance shared across all HID devices and applies a debouncing logic to discard auto-repeat events that would not contribute genuine entropy.

Device identifier injection (`add_device_randomness`). This source is functionally different from the others in that it injects device-specific values such as MAC addresses, serial numbers, and similar identifiers into the input pool while **not contributing to the kernel's entropy estimate**. Its purpose is to personalize the entropy pool, ensuring that two systems with otherwise identical entropy histories, such as freshly cloned VM images, diverge as soon as their distinct device identifiers are mixed in. This addresses one specific class of failure mode that the BSI 2016 study did not yet emphasize.

Hardware RNG framework (`add_hwgenerator_randomness`). This source serves as a generic interface for hardware random number generators exposed via the kernel's hwrng subsystem. Examples include random number generators provided by TPM devices, dedicated USB hardware RNGs, and, in a form directly relevant to cloud environments, `virtio-rng`, the paravirtualized RNG device that hypervisors can expose to guest VMs. The hwrng framework decouples the entropy source from CPU-specific mechanisms, allowing the same code path to handle any conforming hardware generator.

Bootloader entropy (`add_bootloader_randomness`). This source accepts entropy provided by the system firmware or bootloader at startup, most commonly on modern x86 systems through the UEFI Random Number Generator Protocol, which exposes platform-specific entropy to the kernel during early boot. It thereby addresses the boot-time entropy starvation problem identified by the BSI 2016 study, since entropy arriving before the kernel has had time to collect significant interrupt or disk timing data allows the input pool to reach its initial seeding threshold considerably earlier than would otherwise be possible.

VM fork detection (`add_vmfork_randomness`). This source was introduced in kernel 5.18 (2022) and represents one of the most consequential additions to the entropy architecture since the BSI 2016 analysis. It responds to changes in the VM Generation ID, an identifier provided by the hypervisor that changes whenever a VM is forked, cloned, or restored from a snapshot. When such a change is detected, the kernel triggers immediate reseeding of the CRNG, preventing the dangerous failure mode in which multiple VM instances derived from the same snapshot produce identical random number sequences. This source directly addresses the snapshot-related vulnerabilities documented by Ristenpart and Yilek [27].

Auxiliary sources. Several additional sources contribute entropy in more specific or peripheral ways. Scheduler-based entropy collection extracts unpredictability from the timing of context switches and related scheduling events. Power management event collection draws entropy from suspend and resume operations, contributing primarily on mobile and desktop systems rather than servers. The Latent Entropy GCC Plugin is a compile-time mechanism that injects entropy into

TABLE I
ENTROPY SOURCES FEEDING THE INPUT POOL OF THE CURRENT LINUX KERNEL.

Source	Type	Available since	BSI 2016 covered	Entropy Credit
<code>add_interrupt_randomness</code>	Software (timing)	Pre-2016	Yes	Yes
<code>add_disk_randomness</code>	Software (timing)	Pre-2016	Yes	Yes
<code>add_input_randomness</code>	Software (timing)	Pre-2016	Yes	Yes
<code>add_device_randomness</code>	Identifier	Pre-2016	Named, not assessed	No
<code>add_hwgenerator_randomness</code>	Hardware (external)	Pre-2016	Named, not assessed	Yes
<code>add_bootloader_randomness</code>	Lifecycle (boot)	Substantially expanded post-2016	No	Yes
<code>add_vmfork_randomness</code>	Lifecycle (VM fork)	Kernel 5.18 (2022)	No	Triggers reseed
Scheduler-based	Auxiliary	Post-2016	No	Yes
Power management	Auxiliary	Post-2016	No	No
Latent Entropy GCC Plugin	Auxiliary (compile-time)	Post-2016	No	No
RDRAND / RDSEED	Hardware (CPU)	Pre-2016	Mentioned, not central	Yes (with <code>random.trust_cpu=Y</code>)

selected kernel data structures during compilation, providing a source of unpredictability that is independent of runtime events. Each of these contributes modestly to the input pool and is enumerated for completeness, their individual contributions remaining limited compared to the primary sources discussed above.

CPU hardware RNGs. Intel RDRAND and RDSEED, together with their AMD equivalents, are CPU-internal hardware random number generators accessed through dedicated instructions. They are treated separately from the kernel’s general entropy collection functions and have become the most consequential entropy sources in the current Linux RNG architecture, particularly during early boot, where they enable the kernel to reach the initial seeding threshold within milliseconds rather than the 25 to 90 seconds reported in the BSI 2016 study. Because of their central role in this paper, RDRAND and RDSEED are treated in detail in Section 3.2.4.

Summary and comparison to BSI 2016. The following table summarizes the entropy sources in the current Linux RNG and indicates which were present in the architecture analyzed by the BSI 2016 study. Sources marked as new since 2016 represent additions motivated by lessons learned in subsequent years, particularly regarding virtualized environments and the boot-time entropy problem. The table also indicates whether each source receives a kernel entropy credit (contributing to the initial seeding threshold) or is mixed in for personalization purposes only.

The most significant structural change since 2016 is the elevation of the CPU hardware RNGs RDRAND and RDSEED from a peripheral, optional contribution to a central role, particularly during early boot. The addition of `add_vmfork_randomness` and the expanded scope of `add_bootloader_randomness` reflect specific responses to virtualized-environment failure modes identified in the literature. Conversely, the relative contribution of disk-based entropy has declined with the dominance of SSDs, and HID-based entropy is effectively absent in the cloud server deployments that motivate this paper. The source that has undergone the most substantial

change in role, RDRAND together with RDSEED, is examined in detail in the following section.

3.2.4 RDRAND as One Input to the Entropy Pool

Among the entropy sources enumerated in the previous section, RDRAND occupies a special position. It is the only high-throughput hardware entropy source available as a CPU instruction, requiring neither peripheral devices nor a hardware event stream to function. This property is particularly consequential in cloud environments, because the software-derived entropy sources that dominate on physical systems, namely interrupt timing, disk I/O, and HID events, are either absent, weakened, or mediated by the hypervisor in virtualized deployments, as Section 3.2.3 set out. RDRAND, by contrast, executes directly within the CPU and is structurally unaffected by the absence of peripherals, making it the natural candidate to compensate for the entropy deficit of virtual machines. This section describes how RDRAND is integrated into the Linux entropy pool, while the internal architecture of the instruction itself is treated separately in Section 3.3.

Integration into the entropy pool. RDRAND contributes to the Linux RNG through two distinct pathways. During early boot, the kernel queries the CPU’s hardware entropy source directly to seed the input pool before sufficient entropy has accumulated from the slower software sources. Consistent with the intended division of labor between the two instructions established in Section 3.3.2, the kernel prefers RDSEED for this seeding role and falls back to RDRAND only where RDSEED is unavailable, as on pre-Broadwell Intel processors, and where no CPU-based entropy source is present at all it resorts to the CPU’s high-resolution timer to fill the buffer [24, Sec. 3.3.1.3]. During regular operation, the hardware RNG output is mixed into the input pool as one of several contributions during reseeding events. In both cases the output passes through the same BLAKE2s combiner as every other source rather than being given a privileged path that bypasses the combining mechanism [24, Sec. 3.3.1.3]. What distinguishes the CPU hardware RNG from the other sources is not the mechanism of its integration but the quantity and

timeliness of the entropy it can supply, since interrupt-based collection accumulates entropy gradually over time whereas a single sequence of hardware RNG instructions can deliver the full 256-bit initial seed effectively instantaneously.

Entropy crediting. As introduced in Section 3.2.2, whether the kernel credits the CPU hardware RNG output with entropy is governed by the boot parameter `random.trust_cpu`, which defaults to enabled on most modern distributions. The practical significance of this setting is most pronounced during boot. When the CPU source is credited (`trust_cpu=Y`), its output counts toward the 256-bit initial seeding threshold, and the base CRNG can be considered fully seeded almost immediately after the kernel begins execution [24, Sec. 3.3.1.4]. When it is not credited (`trust_cpu=N`), the output is still mixed into the pool but does not advance the entropy estimate, forcing the kernel to reach the seeding threshold using only the other, slower sources [24, Sec. 3.3.1.4]. The choice between these two configurations is a direct expression of the trust an operator places in the CPU hardware RNG, and it has immediate consequences for system behavior, a tension that Chapter 4 examines in detail.

Resolution of the boot-time entropy problem. The boot-time entropy starvation identified by the BSI 2016 study, in which the input pool required a substantial interval after boot to accumulate sufficient entropy [3], is largely resolved in the current architecture wherever the CPU hardware RNG is credited as a boot-time source. Because RDSEED (or RDRAND as its fallback) can supply the full initial seed effectively instantaneously, the window during which cryptographic operations might proceed with insufficient entropy is closed. This represents one of the most significant practical improvements over the state analyzed in 2016. This resolution is nonetheless conditional, holding only under the assumption that the CPU hardware RNG is trusted and credited. The improvement therefore does not eliminate the underlying concern so much as transform it, in that the problem shifts from insufficient entropy at boot to reliance on a single, non-auditable hardware source at the moment of seeding. The quantitative comparison of boot-time entropy behavior between the 2016 findings and the current architecture is presented in Chapter 4.

The structural tension at boot. The combiner guarantee established in Section 3.2.2 states that the output of the input pool is secure as long as at least one contributing source provides sufficient entropy. During steady-state operation on a physical system, this guarantee is robust, because multiple independent sources contribute in parallel. During early boot in a virtualized environment the situation is different, because the software-derived sources have not yet accumulated meaningful entropy, HID sources are absent, and disk-based entropy is weak or mediated. At this moment, the CPU hardware RNG is frequently the only source supplying high-quality entropy. The combiner formally still operates over multiple inputs, yet its security guarantee effectively reduces to the single premise that the CPU hardware RNG is trustworthy. This is precisely the condition under which the trustworthiness of RDRAND and RDSEED becomes most critical, and it is the condition that prevails in the confidential cloud deployments that moti-

vate this paper. To assess whether this reliance is justified, the internal architecture of RDRAND and RDSEED must be understood in detail, which is the subject of the following section.

3.3 Intel RDRAND and RDSEED

3.3.1 Internal Architecture of the Intel DRNG

The hardware behind the RDRAND and RDSEED instructions is a self-contained hardware IP block that Intel markets as Intel Secure Key and that appears in the ESV documentation as the synthesizable RTL component IP-DRNG 4 [21]. The DRNG does not reside inside the individual CPU cores but in the uncore region of the die, the part of the chip that all cores share, and there is exactly one instance per die, so that every core and every hardware thread accessing RDRAND or RDSEED communicates with the same generator over the internal interconnect. This shared placement has a consequence that becomes relevant for the measurements in Chapter 7, because virtual machines drawing random numbers in parallel on the same package compete for a single physical source, which matters in particular for RDSEED throughput under load. Internally, the DRNG implements the complete pipeline introduced in Section 3.1, from the physical noise source through health testing and conditioning to the output DRBG, and this section walks through these stages in order.

Noise source. At the origin of the pipeline sits a feedback-stabilized metastable latch. The circuit consists of two cross-coupled inverters, two logic gates that each feed their output into the other's input, and this arrangement has two stable states that can be read as 0 and 1 together with an unstable equilibrium point exactly between them. A mechanical picture makes the principle intuitive, the circuit behaving like a ball balanced on the crest between two valleys, where the slightest push decides into which valley it rolls. In every clock cycle the DRNG forces the circuit back onto this crest, and what tips it into one of the two stable states is the thermally induced random motion of electrons, a process that is quantum mechanical in origin and therefore not merely hard to predict but non-deterministic in principle. Each settling event yields one raw bit, and because the latch is self-clocked it produces raw bits at gigahertz rates, on the order of three gigabits per second.

A real latch, however, is never perfectly balanced. Manufacturing tolerances and temperature gradients make the two stable states asymmetric, so that an uncorrected circuit might settle into one state in seventy percent of all cycles rather than half of them. The feedback mechanism that gives the design its name counteracts this by continuously observing the output statistics and adjusting the bias voltage of the circuit, pushing the tipping point back toward symmetry whenever the output begins to drift. This correction is effective but never perfect, which is why the raw output carries between 0.6 and 0.7 bits of min-entropy per bit rather than the theoretical maximum of 1.0, as documented in the entropy assessments of the ESV certificates [12], [21]. Removing this residual structure is the task of the conditioning stage, and the raw entropy rate quoted

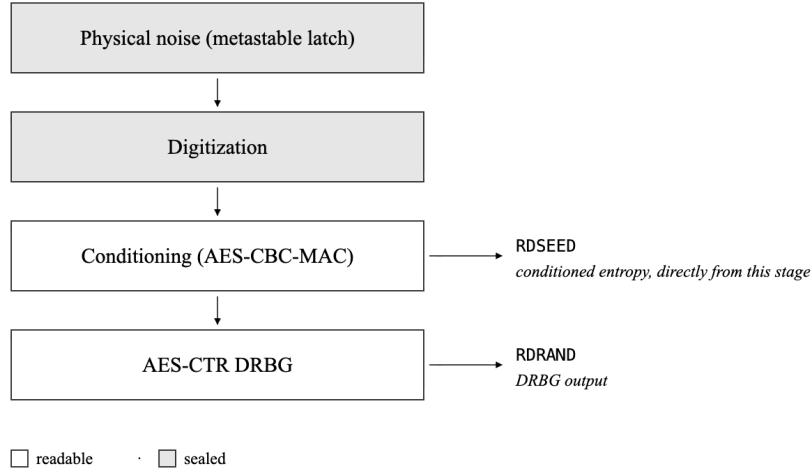


Fig. 2. Intel DRNG pipeline and its two software interfaces

here is precisely the input value on which the conditioning argument in Section 2.2.1 is built.

Health tests. Before any conditioning takes place, the raw bitstream passes through health tests implemented directly in hardware. Their purpose is easily misunderstood and therefore worth stating precisely. Health tests do not measure entropy and do not certify quality, their sole function is to detect a total failure of the physical source immediately. A metastable latch can break through a manufacturing defect, through aging, or through extreme temperature or voltage manipulation, and a broken latch no longer settles randomly but emits, for example, a constant stream of ones or a short repeating pattern. Without a check at this exact point in the pipeline, such a failure would remain invisible forever, because the conditioner and the DRBG downstream would continue to transform the defective input into output that appears statistically flawless while being predictable to anyone who knows the failure mode. The health tests therefore sit at the only place where the defect is still observable, in front of the cryptographic post-processing, and they act as a circuit breaker rather than as a measuring instrument.

The framework for these tests comes from NIST SP 800-90B, which mandates at minimum the Repetition Count Test for detecting a stuck source and the Adaptive Proportion Test for detecting a dominant output value, while explicitly permitting vendor-defined tests provided they detect the same failure modes at least as reliably [11]. Intel makes use of this allowance. The vendor-defined Continuous Health Test evaluates every 256-bit block of raw noise data against expected pattern frequencies and maintains a pass rate over the most recent 256 blocks, corresponding to 65536 bits, declaring the source failed when that rate drops below fifty percent [21]. In addition to the continuous tests, a set of startup tests runs at power-on and after every reset, consisting of a logic integrity built-in self-test that drives deterministic sequences through the entire digital logic and compares the result against the expected output, and a noise source health test that observes a probationary period of 65536 raw bits whose data is dis-

carded in its entirety, all of which completes before the first instruction can be served. During operation, an intermittent test failure raises an internal flag while the generator continues to run, whereas repeated failures cause the DRNG to shut down, reset itself, and re-run the built-in self-test.

Crucially, none of this is observable from software. There exists no instruction, no model-specific register, and no interface of any kind through which the health test status, the pass rates, or the tested raw data could be read out. The only externally visible signal is indirect, in that a failed source causes RDRAND and RDSEED to return with the carry flag cleared and thus to deliver no data at all. The health tests are consequently not a path to the raw entropy values for this study or for any other external party, a limitation whose methodological consequences are taken up in Section 3.3.3 and in Chapter 6.

Conditioning. The raw bits that pass the health tests enter an AES-CBC-MAC conditioner operating at the fixed compression ratio of 512 input bits to 128 output bits. The entropy accounting of this stage was developed in Section 2.2.1 and is only summarized here, in that at a raw rate of 0.6 bits per bit each 512-bit block carries roughly 307 bits of genuine min-entropy, which comfortably exceeds the threshold that the full-entropy requirement places on the conditioner input, so that each 128-bit output block can be treated as carrying approximately one bit of min-entropy per bit.

DRBG. The final stage is an AES-CTR DRBG as specified in NIST SP 800-90A, seeded from the conditioned 128-bit blocks and responsible for producing the fast output stream that consumers actually receive. The reseeding is autonomous, because the hardware does not wait for requests but renews the DRBG state whenever the conditioner has fresh blocks available, and at the production rate of the noise source this happens so frequently that the generator remains far below the maximum reseed interval of 2^{48} Generate calls that the standard would permit.

3.3.2 Differences and Use Cases of RDRAND and RDSEED

The two instructions through which software accesses the DRNG tap the pipeline at different points, and the common shorthand holds that RDRAND delivers DRBG output while RDSEED delivers conditioner output and therefore entropy in a rawer form. For the purposes of this paper the shorthand needs one important refinement. According to the ESV documentation of current silicon, the RDSEED path is implemented as an NRBG construction following draft SP 800-90C, in which the conditioner output is XORed with an output block of the DRBG before it leaves the generator [21], [28]. The full-entropy guarantee of this construction rests entirely on the conditioner component, and the XOR with the DRBG output cannot degrade it, which is an application of the same lemma introduced in Section 2.2.3 with the conditioner output taking the role of the independent, fully entropic input. The refinement nevertheless matters, because it means that RDSEED also delivers cryptographically post-processed bits rather than the raw conditioner stream, so that both instructions expose only what Section 2.2.2 called the shadow of the entropy source. Neither of them offers a view of the raw physical process, a fact whose consequences for the interpretation of the measurements are addressed in Chapter 6.

The practical differences between the instructions follow directly from their positions in the pipeline. RDRAND draws from the DRBG, which always has output ready, and is therefore fast and practically inexhaustible. RDSEED is bound to the physical replenishment rate of the conditioner and can run empty when demand exceeds the pace at which the noise source delivers fresh raw blocks, a situation that arises in practice under sustained parallel load. Both instructions signal their outcome through the carry flag, where CF set to 1 indicates that the destination register holds a valid value and CF set to 0 indicates that no data was available and the call should be retried. Intel recommends a bounded retry loop for RDRAND, whose failures are rare and transient, while callers of RDSEED must be prepared for longer waiting periods between attempts, since its availability depends on physical entropy accumulation rather than on algorithmic readiness.

The intended use cases mirror this division. RDSEED exists to seed other random bit generators, for instance the DRBG instances of an operating system or a cryptographic library, where full-entropy input is required and throughput is secondary. RDRAND exists for the running random number demand of applications, where speed matters and DRBG output is exactly what is needed. A final operational property is relevant for the experimental chapters of this paper, since a single DRNG instance per package serves all cores, so that multiple virtual machines measuring simultaneously on the same host draw from the same physical source, and contention effects, particularly RDSEED underflow under load, must be accounted for in the measurement design in Chapter 6.

3.3.3 Vendor Opacity and the Limits of Auditability

The criticisms that have accompanied RDRAND since its introduction operate on three distinct levels that are frequently conflated and are therefore treated separately here.

The first level is architectural opacity as a deliberate design decision. Raw data access is disabled in hardware, debug ports are cut off at the security boundary, and no configuration of the DRNG is possible from software at any privilege level [21]. From Intel's perspective these properties are security features, because a generator whose internals cannot be reached cannot be manipulated or leaked through the same channels. From the perspective of an external evaluator, however, the identical properties are the reason why no independent party has ever measured the raw noise source of a production CPU. The only external in-depth analysis that exists, the report by Cryptography Research on the Ivy Bridge DRNG from 2012, was conducted under a non-disclosure agreement and on the basis of test data that Intel itself provided [29], which makes the report both the best available external evidence and an illustration of the very access problem it could not overcome.

The second level concerns manipulability below the detection threshold, and here the work of Becker et al. on dopant-level hardware trojans is the central reference [30]. The authors demonstrated that a TRNG of exactly this construction can be sabotaged by altering the dopant polarity of individual transistors, in a way that drastically reduces the delivered entropy while leaving the circuit layout untouched, so that optical inspection reveals nothing and even the built-in health tests continue to pass. This result connects directly to the principle established in Section 3.1, namely that the output of a correctly implemented DRBG looks statistically perfect regardless of the state of the source behind it, and it constitutes the strongest technical argument for why statistical inconspicuousness of RDRAND output can never by itself prove that the underlying source is intact.

The third level is the historical trust context in which these technical concerns are read. The Snowden disclosures of 2013 documented the deliberate weakening of the NIST-standardized Dual EC DRBG, establishing that the concern about vendor-introduced backdoors in randomness infrastructure is not paranoia but precedent. In the same year, a public petition demanded the removal of RDRAND from the Linux kernel [31], and the debate that followed ended in precisely the compromise described in Section 3.2.2, under which RDRAND is never used exclusively but only as one input among several to the entropy combiner, so that its potential compromise is contained by the independence of the remaining sources. Whether and how far these criticisms are mitigated by the validation evidence discussed in Section 2.3.1 and by the architectural changes that confidential computing introduces is examined in Chapter 5, and the present section deliberately confines itself to the technical substance of the criticism itself.

3.4 AMD RDRAND and RDSEED

3.4.1 The Publicly Documented AMD Architecture

AMD added support for RDRAND in 2015, with reliable operation established from the Zen generation onward, and introduced RDSEED alongside the same processor generations. The underlying entropy source departs from Intel's design at the physical level, since AMD relies not on a single metastable latch but on an array of sixteen ring oscillators running in parallel. A ring oscillator is a chain formed from an odd number of inverters connected in a loop, so that the signal constantly flips between 0 and 1 and free-runs at a characteristic frequency that depends on the propagation delay of each inverter. Randomness enters through jitter, the minute thermally induced variation in the exact moment at which each flip occurs, which is converted into bits by sampling the oscillating signal against a reference clock. Because sixteen such rings run side by side, a single sampling step already yields a 16-bit raw output rather than the single raw bit that Intel's latch produces per cycle.

The health testing built around this source follows the same two basic test types already introduced for Intel, the Adaptive Proportion Test and the Repetition Count Test, both running at start-up and continuously during operation. AMD's public documentation describes a failure mode specific to its own architecture, in that individual rings can stop oscillating, which freezes the corresponding bits at a constant value and makes certain output symbols occur far more often than chance would allow, a distortion that the Adaptive Proportion Test is virtually guaranteed to catch.

Where the AMD design diverges most visibly from Intel's is in what software is permitted to see. Access to the unconditioned raw data is not merely possible but an official and documented part of the interface, since software can read the sixteen-bit raw samples of the ring oscillators directly through the read-only TRNG_RAW register, reachable through the memory-mapped I/O space of the AMD Secure Processor. AMD goes a step further and documents how an end user can verify the source's quality independently, by collecting one million sixteen-bit raw samples and evaluating them with NIST's `ea_non_iid` tool. This openness at the raw-data stage stands in deliberate contrast to Intel's design, where no equivalent interface exists and the noise source is entirely sealed within the silicon.

The certification picture, however, introduces a complication that requires careful interpretation before the reported entropy values can be read correctly. AMD's conditioner is classified as non-vetted under the terminology of SP 800-90B [32], which means it does not carry the formal guarantee that Intel's AES-CBC-MAC conditioner receives as a vetted component. A vetted conditioner permits the entropy of its output to be derived from the construction itself, so that as long as the input contains sufficient entropy the output may be treated as fully entropic without direct measurement. Because AMD's conditioner lacks this guarantee, the entropy of each RDSEED block must instead be established empirically for each product family through SP 800-90B testing and reported as a mea-

sured value. The certified entropy values for AMD and Intel therefore do not compare like with like: Intel's value is a construction-based assertion of full entropy per bit of output, while AMD's values are empirically measured quantities that reflect what the output stream actually carries, expressed as bits of min-entropy per 128-bit RDSEED block rather than per individual bit.

Reading the AMD ESV certificate with this framing in mind, the numbers become interpretable. The EPYC 7xx2/7xx3 family carries 1.31 bits of min-entropy per 128-bit RDSEED block, and the Ryzen V1xxxx family carries 0.656 bits per block [32]. A single RDSEED call on EPYC hardware therefore delivers only 1.31 bits of genuine unpredictability, not the 128 bits of genuine entropy that Intel's fully conditioned output provides per call. This is not a sign that the ring oscillators are fundamentally weaker as a physical source, but a consequence of how the certification methodology accounts for a non-vetted conditioner. Because those values are measured for each product family rather than derived from the construction, they also differ considerably between families. The same table certifies seven of them, reaching from 0.656 bits per block for Ryzen V1xxxx at the low end to 1.63 bits for Ryzen V3xxx at the high end, with EPYC 9xx4/8xx4 at 1.07 bits and EPYC 3xxx at 0.796 bits lying between, so that the entropy a single RDSEED call delivers varies by a factor of roughly two and a half depending on which silicon executes it [32]. No single figure can therefore stand for AMD hardware in the way that Intel's construction-based claim stands for every part carrying the vetted conditioner, which is one reason the measurement design in Section 6.2.5 records the processor family of each platform rather than its marketed product name.

A particularly revealing detail in the AMD ESV certificate reinforces this interpretation. The min-entropy of the raw 16-bit ring oscillator output is almost identical to the min-entropy of the conditioned RDSEED output, since for EPYC 7xx2/7xx3 the raw value is 1.31225 bits per 16-bit sample against 1.31221 bits per 128-bit RDSEED block. This near-identity reveals that AMD's conditioner is not performing entropy extraction in the way a vetted conditioner would, compressing many low-entropy raw bits into fewer high-entropy output bits. Instead, the conditioned output carries essentially the same amount of entropy as a single raw sample, packaged differently, which means the entropy available per RDSEED call is bounded by the entropy of one ring oscillator sample rather than accumulating across many raw bits as one might expect from a conditioning stage.

The practical consequence for any system seeding a downstream DRBG from RDSEED on AMD hardware is that multiple calls must be accumulated until the total entropy income reaches the required security threshold. For the EPYC 7xx2/7xx3 family, reaching 256-bit security requires 196 RDSEED calls when a nonce is provided, or 293 calls when a random nonce is required [32], against the single call that suffices on Intel hardware where the vetted conditioner delivers full entropy per block. For the Ryzen V1xxxx family the requirement rises to 391 or 586 calls respectively because the per-block entropy is lower at 0.656 bits per 128-bit block.

This is not a security vulnerability in itself, since the entropy is genuinely there and can be accumulated, but it is an operational constraint with real consequences for systems that call RDSEED only once and assume they have received sufficient entropy to seed a cryptographic key. The AMD ESV certificate makes this requirement explicit by specifying exactly how many blocks must be collected for each security strength and product family, which means a correctly implemented system can satisfy BSI and NIST requirements for entropy quality on AMD hardware provided the accumulation requirement is respected.

For RDRAND itself, the public documentation thins out considerably. AMD’s ESV public use document describes RDSEED and the raw-data path in detail but does not mention RDRAND at all, and the AMD64 Architecture Programmer’s Manual documents RDRAND only at the level of observable behavior [33], covering its opcode, the CPUID feature bit that announces its availability, and the carry-flag semantics that signal success or failure. What remains undocumented is the internal hardware construction that produces this behavior, so that it is publicly unclear whether RDRAND on AMD silicon is served by a dedicated hardware DRBG seeded from the ring-oscillator output in the manner of Intel’s design, or whether it is implemented in closer technical proximity to RDSEED than its name would suggest. A further data point on the reliability of the public record concerns early AMD silicon predating Zen, where on processor families 15h and 16h RDRAND was demonstrably prone to failure after resuming from suspend due to a missing BIOS initialization step, a problem documented in Linux kernel mailing list discussions and ultimately addressed by a kernel patch that clears the RDRAND CPUID bit on affected systems unless the user explicitly forces its use through the `rdrand=force` parameter.

A more recent and more consequential instance of generation-specific behavior concerns RDSEED on fifth-generation AMD EPYC processors, the 9005 series based on the Zen 5 microarchitecture, where a hardware defect in the 16-bit and 32-bit RDSEED variants was publicly confirmed in late 2025 as CVE-2025-62626 [34]. Under sustained memory pressure, these variants can return the value zero while simultaneously asserting the carry flag as success, meaning software has no mechanism to detect the failure and may use a predictable value as entropy input. The 64-bit RDSEED variant is unaffected. The defect was discovered by engineers at Meta and reported on the Linux kernel mailing list in October 2025, leading to a Linux kernel patch that disables RDSEED on affected Zen 5 platforms entirely. AMD rated the defect as high severity and released AGESA microcode updates from mid-November 2025 for the EPYC 9005 series, with the embedded variants following through January 2026. The defect is directly relevant to this paper in two respects, since it demonstrates empirically that the hardware implementation of RDSEED differs in security-relevant ways across CPU generations, and since it imposes a concrete methodological requirement on the measurements in Chapter 6, namely that the generation and microcode patch level of any AMD platform used for measurement must be recorded before RDSEED results can be

interpreted at all. That record comes from the manifest written alongside each measurement, which states family, model, stepping and microcode revision. The AMD platform used here reports family 25, model 17, stepping 1 and microcode revision 0xa101158, running under platform security firmware reporting SEV API 1.58 build 2. For this platform the question is therefore settled by the generation alone, because family 25 is Zen 4 and the defect is confined to the Zen 5 generation. Inside a confidential guest the attestation report would add the TCB version under which the guest was launched, but no report could be obtained in the environment measured here, for the reasons Section 6.2 records.

3.4.2 Differences to the Intel Implementation

Read against Section 3.3, the AMD design does not merely vary Intel’s approach in detail but inverts one of its central design choices, and it is this inversion, rather than a restatement of either architecture, that this section examines. The comparison rests on three points of contrast, each of which follows from a different design decision. Fig. 3 places the two pipelines side by side, and the inversion is visible in where each vendor opens its design to inspection.

The first is where each vendor chose to place transparency within the pipeline. Intel documents its complete architecture down to the DRBG construction yet seals every point of data access, so that the only externally visible fact about the noise source is that it functions or does not. AMD makes the opposite trade, exposing the raw output of its noise source through a documented register and even instructing end users on how to test it themselves, while leaving the internal construction of RDRAND, its best-known and most-used instruction, essentially undocumented. Neither vendor is more transparent than the other in any simple sense, each has instead opened a different stage of the same pipeline to inspection, which means that an external verifier faces a different evidentiary gap depending on which manufacturer’s silicon is being examined. For the opacity argument developed in Section 3.3.3, this comparison carries the specific consequence of demonstrating that sealing the raw data path is not a technical necessity of CPU-integrated entropy sources, since a viable production design exists that does the opposite, which reframes Intel’s choice as a deliberate decision rather than an inevitable one.

The second point of contrast concerns what the certified numbers are actually allowed to mean. Because Intel’s conditioner is vetted, its certified output can be treated as carrying full entropy across the board, and the entropy accounting in Section 2.2.1 could rely on a single, stable assumption. Because AMD’s conditioner is not vetted, no such blanket assumption is available, and the entropy credited to RDSEED output instead varies by product family, sitting far below Intel’s raw entropy rate in the concrete cases cited in Section 3.4.1. This is not a difference in numerical detail but a difference in what kind of claim the certification supports, and it means that a statement such as “the output carries full entropy” cannot be transferred from the Intel case to the AMD case without first checking which processor family is in use,

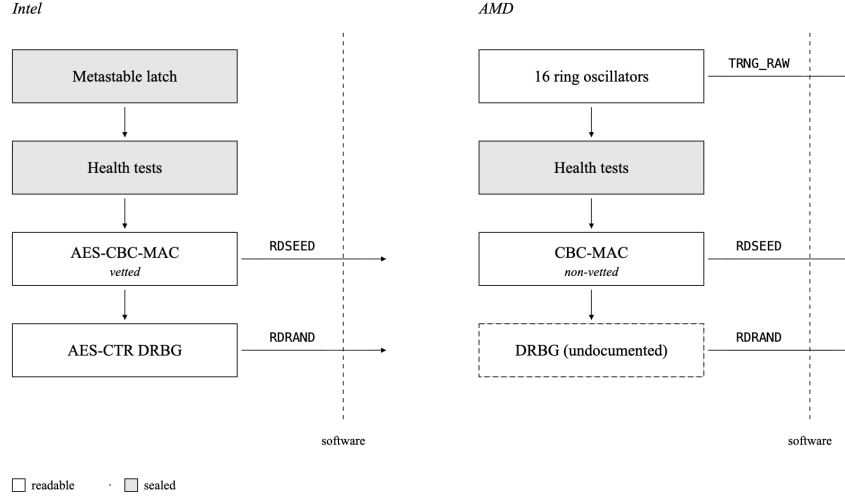


Fig. 3. Side-by-side comparison of the Intel and AMD DRNG pipelines

a distinction the measurement design in Chapter 6 needs to preserve wherever both platforms are compared.

The third point of contrast is that the two architectures leave behind evidentiary gaps of different shapes. Intel’s gap is a closed system that is thoroughly documented on paper but physically inaccessible for independent verification, while AMD’s gap is an open data path attached to an instruction, RDRAND, whose internal behavior is not documented at all. Neither gap is smaller than the other, but they would need to be closed by different means, physical access to silicon in Intel’s case and vendor disclosure of the RDRAND construction in AMD’s case, a distinction that matters for how the trust question is framed separately for each platform in Chapter 5.

4 VIRTUAL MACHINE MONITOR IMPACT ON NOISE SOURCES

4.1 KVM/QEMU in Cloud Virtualization Architecture

Understanding how virtual machines are hosted in modern cloud environments is a prerequisite for analyzing how entropy sources behave within them, because the virtualization stack determines what the guest operating system can see, what the hypervisor can intercept, and where the boundaries of trust are drawn.

The dominant open-source virtualization stack on Linux servers combines two components that together form a complete hypervisor. KVM, short for Kernel-based Virtual Machine, is a module that loads directly into the Linux kernel and activates the hardware virtualization extensions built into modern processors, called VT-x on Intel platforms and AMD-V on AMD platforms. These extensions are standard features integrated into virtually every server-class processor produced since 2006, enabling the processor to switch between a privileged hypervisor mode and a controlled guest mode

in hardware rather than software. KVM itself handles CPU scheduling across virtual machines and enforces memory isolation between them, but it does not emulate any devices on its own. That role falls to QEMU, the Quick Emulator, a userspace program that presents each guest with a complete set of virtual hardware including network adapters, storage controllers, and other peripherals, intercepting guest accesses to these virtual devices and translating them into operations on the actual host hardware so that the guest believes it is running on dedicated physical machines while all device communication is mediated in software. When a Linux server starts a virtual machine, KVM and QEMU always operate together, with KVM providing hardware-accelerated CPU and memory virtualization and QEMU providing the surrounding device environment. The two together constitute what practitioners and this paper refer to as the KVM/QEMU hypervisor stack.

This stack is not merely an academic reference implementation. Two of the three largest cloud providers by market share build their guest compute infrastructure on KVM as a foundation. Google Cloud Platform runs its Compute Engine virtual machines on a security-hardened KVM-based hypervisor and contributes substantially to the upstream KVM project [35]. Amazon Web Services replaced its earlier Xen-based infrastructure with the Nitro System beginning in 2017 [36], which centers on a stripped-down KVM-derived hypervisor that offloads device emulation to dedicated Nitro Cards rather than running it in software, reducing the hypervisor attack surface while retaining KVM’s CPU and memory isolation model. Microsoft Azure uses a hypervisor derived from Hyper-V rather than KVM, making it an outlier among the three major providers and outside the scope of this paper’s empirical analysis. The focus on KVM/QEMU therefore reflects where the majority of cloud workloads actually run rather than an arbitrary laboratory choice.

In a typical cloud deployment a single physical server runs many guest virtual machines simultaneously, each isolated from the others through the memory and CPU partitioning that KVM enforces. From the perspective of each guest the

hardware appears dedicated in a sense that the guest operating system believes it has exclusive access to its own processor, its own memory, and its own devices, while in reality all guests on the same physical host share the underlying hardware coordinated invisibly by the hypervisor. The architectural consequence of this arrangement is that the hypervisor occupies a uniquely privileged position, sitting below every guest operating system in the software stack with full visibility into guest memory and the ability to intercept any guest action that requires mediation. This includes not only device accesses but, under certain configurations, individual CPU instructions.

The relationship between the hypervisor's privileged position and entropy generation is the central concern of this chapter. A guest operating system that calls RDRAND is issuing a CPU instruction that under normal circumstances executes directly on the hardware and returns a value without involving the hypervisor, making it fast, independent of the guest's device environment, and in principle unaffected by the virtualization layer. Whether this independence holds in practice, and under what conditions it can be violated, is what the following sections examine. The answer depends on how VM-exits are configured for RDRAND on each platform, how that configuration differs between Intel and AMD, and whether the architectural changes introduced since the BSI 2016 study have altered the attack surface in any meaningful way.

4.2 How VMMs Intercept CPU Instructions

Chapter 3 described the entropy sources available to a Linux system and the role that RDRAND and RDSEED play among them. Assessing how a virtual machine monitor can affect these sources first requires an account of the mechanisms by which a VMM observes, intercepts, and alters the execution of instructions issued by a guest virtual machine. This section describes those mechanisms neutrally, as general capabilities of the virtualization layer, while their concrete application to RDRAND in the interception attack is treated in Section 4.3.

Two properties of the x86 execution model underpin the discussion that follows. First, code executes at one of several privilege levels. Ordinary application and operating-system code in a guest runs at a reduced privilege, which conceptually means that guest code executes in the processor's non-root operation mode introduced by hardware virtualization while the VMM operates at the higher-privileged root mode. Second, certain events cause the processor to suspend the guest and transfer control to the VMM, a pattern historically known as trap-and-emulate. The guest attempts a sensitive operation, the processor traps it, and the VMM emulates its effect before returning control. Modern hardware-assisted virtualization refines this pattern into a well-defined mechanism, the VM-exit, described below. An older class of techniques based on full software emulation and binary rewriting achieves the same control by different means and is discussed in Section 4.2.2.

4.2.1 VM-Exit Behavior on Intel and AMD

A VM-exit is a hardware-supported, controlled transfer of execution from the guest to the VMM. When a configured triggering condition occurs, the processor suspends guest

execution, saves the guest's architectural state, and transfers control to the VMM. Once the VMM has handled the event, it issues the reverse transition, a VM-entry, which restores the guest state and resumes execution at the point of interruption. The set of conditions that trigger a VM-exit is not fixed. It is configured by the VMM through a dedicated control structure, which allows the VMM to decide precisely which guest operations it wishes to intercept and which it will allow to execute natively on the physical CPU.

On Intel platforms (VT-x), this configuration is held in the Virtual Machine Control Structure (VMCS), a per-virtual-CPU data structure managed by the VMM. Among its many fields, the VMCS contains execution control bits that enable or disable exiting for specific instructions and events [37]. Relevant to this work, the VMCS provides a dedicated "RDRAND exiting" control bit and an analogous "RDSEED exiting" bit [37, Sec. 25.6.2]. When the RDRAND exiting bit is set, every execution of the RDRAND instruction by the guest triggers a VM-exit rather than returning a value from the hardware. Upon the exit, the processor records the reason for the transition and associated information in dedicated VMCS fields, namely the exit reason and, where applicable, the exit qualification [37, Sec. 24.9], so that the VMM can determine what the guest was attempting and respond accordingly. That the architecture provides a purpose-built control bit for RDRAND deserves emphasis. Interception of the instruction does not depend on some broader mechanism catching it incidentally, since the platform offers it explicitly as a first-class capability.

On AMD platforms (AMD-V, also referred to as SVM), the equivalent configuration resides in the Virtual Machine Control Block (VMCB), which is functionally analogous to Intel's VMCS. The VMCB holds intercept bits that determine which instructions and events are trapped to the VMM, and the corresponding control transfer is denoted #VMEXIT [38]. The intercept vectors cover a wide range of instructions and events, though not, as Section 4.3.3 sets out in detail, RDRAND or RDSEED. Across the transition, the guest state is saved to and restored from a defined state-save area within the VMCB.

Both architectures therefore expose the same fundamental capability, namely hardware-supported trapping of configurable instructions, and they differ in terminology, with VMCS against VMCB and VM-exit against #VMEXIT, as well as in the internal layout and semantics of the control structures, rather than in principle. Which instructions each platform makes interceptable is a separate question from whether the mechanism exists, and on exactly that question the two diverge for the CPU hardware random number instructions, an asymmetry that Section 4.3 examines in detail and that bears directly on the second research question. It is this divergence, rather than any assumed symmetry, that makes it necessary to examine Intel and AMD platforms side by side instead of restricting the analysis to a single vendor, since any claim about the trustworthiness of RDRAND under virtualization must hold across both dominant server architectures if it is to be meaningful.

One further property of VM-exits bears on the analysis in later sections, namely that they are computationally expensive. Each exit entails saving the guest state, switching context to the VMM, executing the handler, and restoring the guest state on entry. The resulting overhead is considerable, and the literature places it on the order of hundreds to thousands of CPU cycles per transition [39]. Because that figure predates the platforms examined here, the cost of an exit was measured directly for this work on a client processor, an Intel Core i7-1255U. The instruction used to provoke the exit is CPUID, which always exits, and the timing overhead of the measurement itself is subtracted. On the host the instruction costs 110 cycles, and inside a guest it costs 2,994, so an exit adds about 2,880 cycles, which at the timestamp frequency of that part is close to 1.1 microseconds. The order of magnitude rather than the absolute figure is what carries over to a server part, and the measurement confirms the upper end of the range the literature gives. For an instruction such as RDRAND, which an unmodified guest may execute at a high rate, configuring an exit on every invocation imposes a substantial performance cost on the guest. This cost does not prevent interception, but it does bear on the question of whether continuous, wholesale interception of RDRAND is practical to sustain, a point revisited when the interception attack itself is examined in Section 4.3.

4.2.2 Full Instruction Emulation and Binary Rewriting

The VM-exit mechanism described above depends on hardware support that has been available on x86 processors only since the introduction of VT-x and AMD-V in 2006. Before these extensions existed, virtualizing x86 was considerably harder, and the techniques developed to solve that problem remain relevant to the present analysis, since they represent an alternative path by which a VMM can control what a guest observes.

The difficulty that motivated these techniques lies in the design of the x86 instruction set itself. Classical trap-and-emulate virtualization requires that every instruction whose behavior depends on the current privilege level also traps when executed at reduced privilege, so that the VMM can intercept it. The x86 architecture did not satisfy this requirement. It contained instructions that behave differently depending on privilege level yet fail silently rather than trapping when a guest kernel executes them at reduced privilege, returning an incorrect result without raising any exception. The POPF instruction is the canonical example [40], [41]. Because such instructions produce wrong values invisibly, a VMM relying purely on trapping could not guarantee correct guest behavior, and x86 was for many years considered not classically virtualizable.

Two techniques emerged in response. Full instruction emulation abandons native execution entirely for the affected code. Rather than allowing the processor to execute a guest instruction, the VMM decodes it in software, computes its effect itself, and writes the result into the guest's saved state. Because the VMM performs the computation, it determines the outcome completely. Applied to RDRAND, an emulating

VMM would never invoke the hardware instruction at all, and the value the guest receives would be whatever the emulator chose to produce. The cost of this approach is severe, since interpreting every instruction in software is orders of magnitude slower than native execution, which is why full emulation is used selectively rather than for entire workloads.

Binary rewriting, sometimes called binary translation, takes a different route to the same end. Instead of interpreting instructions as they execute, the VMM inspects guest code before it runs and replaces problematic instruction sequences with substitutes of its own choosing, which then execute natively. This was the approach taken by early VMware products to work around the non-virtualizable instructions of pre-VT-x x86, and it achieves control at translation time rather than execution time [42]. The same mechanism that allows a rewriter to substitute a safe sequence for a problematic one would equally allow it to substitute a controlled value for a call to RDRAND.

Both techniques have been largely displaced in production cloud infrastructure by hardware-assisted virtualization, which delivers the same control at a fraction of the performance cost. They are therefore unlikely to be encountered in a modern KVM deployment. What makes them relevant here is that they demonstrate the breadth of the capability under discussion. A VMM that wishes to determine what value a guest receives from RDRAND does not depend on the VM-exit controls examined in Section 4.2.1, which offer no more than the most convenient path among several. Interception is made possible in the first place by the underlying position of the hypervisor, which holds complete authority over the guest's execution and architectural state, and that position is independent of which technique is used to exercise it. This observation matters for the trust analysis in Chapter 5, since it establishes that closing a single mechanism would not be sufficient. Confidential computing must remove the hypervisor's underlying authority over guest state, not merely one interception technique.

4.3 VMM Impact on RDRAND and RDSEED

4.3.1 Reproducing the BSI 2016 Interception Attack

The capability for a VMM to intercept RDRAND, established in general terms in Section 4.2, was concretely demonstrated by the BSI 2016 study through a proof-of-concept implementation in KVM. One of the study's authors implemented a modification to the KVM kernel module that intercepts every RDRAND call issued by a guest and returns zero in its place, with a correctly set carry flag indicating success. The implementation required approximately ten lines of C code modifying two locations in the KVM source. The first modification was in the function that configures the VM-execution controls when creating a virtual CPU, where the RDRAND exiting bit was enabled so that every RDRAND instruction issued by the guest triggers a VM-exit. The second modification was in the existing VM-exit handler for RDRAND, which KVM already contains as part of its standard instruction emulation infrastructure. Rather than forwarding the value returned by the physical hardware, the modified handler writes

zero into the guest’s target register before returning control with the carry flag set to indicate a successful read.

The attack requires root-level access to the host and the ability to compile and load a modified kernel module, placing it firmly within the capabilities of a cloud provider or any party with privileged access to the hypervisor, rather than within those of an ordinary tenant. Once active, the manipulation is undetectable by the guest on a per-call basis. RDRAND exposes no checksum, timestamp, or any other verifiable property through which a caller could distinguish a value produced by the hardware entropy source from one written by a handler. The only observable signal is the carry flag, which the attacker controls directly, and which the guest is architecturally required to treat as the authoritative indicator of success.

Detection is possible only across calls, and the reach of that observation is narrow. A Linux guest applies a self-test to the instruction during early boot, in `x86_init_rdrand`, which draws eight values and requires at least five of them to differ from their predecessor [43]. An interception that substitutes one constant for every call fails that test, so it is caught by an ordinary distribution rather than by anything the tenant has to arrange. The consequence of failing it has hardened over time. The guest observed in this work ran a kernel that only issued a warning and carried on, whereas the current kernel clears the feature bits for RDRAND and RDSEED together and reports that the instruction is not reliable on this platform, which withdraws the source from the guest entirely.

That hardening has a consequence worth stating for confidential guests, although it is reasoned rather than measured. Section 5.5.2 describes a kernel that aborts the boot of a confidential guest when the mandated contribution from the instruction cannot be obtained. A constant-value interception against such a guest on a current kernel would therefore not go unnoticed, it would prevent the guest from starting. The chain holds on the specifications and the kernel source, and it could not be exercised here, because AMD offers no intercept to mount it with and no Trust Domain could be created on the Intel platform available.

The same constancy becomes visible again when values are compared across reboots, where identical outputs appear as duplicates in a restart matrix. Neither observation helps against a manipulation that returns varying values from a source the guest cannot inspect, which is the case the entropy assessment in Chapter 6 is directed at.

The BSI study examined RDRAND exclusively and left the behavior of RDSEED under interception as an open question. As noted in Section 4.2.1, the Intel VM-execution controls provide separate and independently configurable intercept bits for the two instructions [37, Sec. 25.6.2], so the applicability of the attack to RDSEED cannot be assumed from the RDRAND result alone. Whether the attack remains feasible on current KVM versions and current CPU generations, given the substantial changes to both since 2016, and whether RDSEED exhibits the same vulnerability, are questions this paper takes up in Section 7.3. The first is answered there

and the second is not, since the two instructions are trapped through independent control bits and the measurement that would settle the RDSEED case could not be carried out.

4.3.2 VM-Exit Configuration on Intel

The architectural basis for the interception attack on Intel platforms is the two-level structure of the VM-execution controls within the VMCS. The Primary Processor-Based VM-Execution Controls govern fundamental virtualization behaviors and serve as a gateway to a second control word. Their Bit 31, labeled “Activate secondary controls”, must be set before the processor consults the Secondary Processor-Based VM-Execution Controls at all [37, Sec. 25.6.2]. Intel introduced this two-level design because the original VMX specification exhausted the available bits in the primary control word as new features were added, and extending it through a secondary field avoided redefining existing bits or breaking backwards compatibility. In a standard KVM deployment, Bit 31 is already set because KVM relies on secondary controls for a range of other features unrelated to RDRAND, which means the attack requires activating only one additional bit rather than enabling the secondary control layer from scratch.

Within the Secondary Processor-Based VM-Execution Controls, Bit 11 is designated “RDRAND exiting” and Bit 16 is designated “RDSEED exiting” [37, Tab. 25-7]. When Bit 11 is set, every execution of RDRAND by the guest causes a VM-exit rather than returning a value from the hardware entropy source. Bit 16 provides the same capability independently for RDSEED. The two bits are fully orthogonal, so a VMM can intercept RDRAND while leaving RDSEED to execute natively, or vice versa, without any coupling between the two decisions. Both bits are unset by default in a standard KVM deployment, meaning RDRAND and RDSEED pass through to the hardware under normal conditions and the interception requires an explicit configuration change.

The VMCS is a per-virtual-CPU data structure that resides in host memory and is accessible only through the privileged Intel instructions VMREAD and VMWRITE, which are available exclusively in VMX root mode. No userspace process or guest kernel can read or write the VMCS directly. Any modification requires execution in kernel context, either as part of the hypervisor itself or through a kernel module. In KVM, the relevant configuration points are the functions `vmx_exec_control` and `vmx_secondary_exec_control`, which set the primary and secondary control fields respectively when a virtual CPU is created [44]. The BSI proof-of-concept modified `vmx_secondary_exec_control` to include the RDRAND exiting bit, and modified the existing RDRAND VM-exit handler located in `arch/x86/kvm/vmx/vmx.c` and the associated emulation logic in `arch/x86/kvm/emulate.c` to substitute a fabricated value for the hardware output.

The flow that follows a triggered VM-exit illustrates why so little additional code is needed. When the guest executes RDRAND and the exit fires, the processor saves the complete guest architectural state, including all general-purpose registers, into the VMCS state-save area and transfers control to KVM’s exit dispatch table. KVM reads the exit reason from

the VMCS exit-information fields, identifies the instruction that caused the exit, determines which guest register was designated as the target operand, and writes the response value into the corresponding field of the state-save area. On VM-entry, the processor restores the guest state from that area, so the guest resumes with the fabricated value already in its target register and the carry flag set to indicate success, with no observable difference from a native hardware response. This reliance on the standard VMCS save-and-restore infrastructure is precisely what allows the attack to be expressed in so few lines. KVM and the CPU together provide all the scaffolding and only the value written into the target register changes.

The relevant conclusion for the trust analysis is that the interception capability is not incidental to KVM or correctable by patching KVM. It is a first-class feature of the Intel VMX architecture, documented explicitly in the Intel SDM Volume 3C among the definitions of the Secondary Processor-Based VM-Execution Controls, in Section 25.6.2 and Table 25-7 of current revisions, numbered Section 24.6.2 and Table 24-7 before the chapter renumbering. A cloud provider operating a standard KVM stack on Intel hardware possesses the interception capability as a matter of architectural fact, and acquiring it requires no more than enabling two bits in the VM-execution controls.

What those two bits do and do not accomplish has to be kept apart, because the distinction decides how much code an attack actually needs. Setting the bit produces the exit and therefore the interception, but it does not by itself produce a chosen value. Unmodified KVM routes both the RDRAND and the RDSEED exit reason to `kvm_handle_invalid_op` [44], so a guest whose instruction has been trapped receives an invalid opcode exception rather than a fabricated random number, which is a failure mode a caller notices immediately. Standard KVM reaches that path only where the instruction has been hidden from the guest through CPUID, since the exiting control is otherwise cleared whenever the guest is meant to have the instruction at all. Returning a chosen value therefore requires the second modification as well, namely the altered handler that writes into the guest's target register, which is the roughly ten lines the BSI proof-of-concept added and which the reproduction in this work follows. The register to write into is not guessed but read from the VM-exit instruction-information field, whose layout for RDRAND and RDSEED the SDM gives explicitly [37, Tab. 30-12].

4.3.3 VM-Exit Behavior on AMD

On AMD platforms the equivalent of the Intel VMCS is the Virtual Machine Control Block, or VMCB, a per-virtual-CPU data structure that the hypervisor uses to configure guest behavior and that the processor uses to save and restore guest state across transitions. The VMCB is organized into three regions with distinct roles. The control area sits at the beginning of the structure and contains all configuration bits that govern guest execution, including the intercept vectors that determine which instructions and events trigger a VMEXIT. The state-save area holds the complete architectural state of the guest, encompassing all general-purpose registers, segment descrip-

tors, control registers, and related fields, which the processor saves automatically on every VMEXIT and restores on every VMRUN. The exit area records the reason for the most recent VMEXIT and associated context information, such as which instruction triggered the exit and what address was referenced, allowing the hypervisor handler to determine what the guest was attempting. If an AMD hypervisor were to intercept RDRAND, the manipulated return value would be written into the state-save area before resuming the guest, in the same way that KVM writes into the VMCS state-save area on Intel platforms.

The critical difference from Intel lies in what the VMCB control area actually offers. The intercept vectors in the VMCB control area are documented in AMD64 Architecture Programmer's Manual Volume 2, Section 15.9, whose Table 15-7 lists every instruction and event for which AMD SVM provides a dedicated intercept bit [38, Tab. 15-7]. This list covers a wide range of operations including CPUID, RDTSC, RDTSCP, HLT, and various privileged operations, but it does not include RDRAND or RDSEED. There are no VMCB control bits labeled "RDRAND intercept" or "RDSEED intercept" in any revision of the AMD64 manual. The BSI 2016 study drew its conclusion from exactly this observation [3, Sec. 4.3.2]. Because no dedicated intercept mechanism exists for these instructions, an AMD hypervisor operating through the standard SVM interface cannot configure a VMEXIT on RDRAND in the way that an Intel hypervisor can by setting the RDRAND exiting bit. AMD has extended the VMCB intercept vectors in subsequent manual revisions to cover additional instructions, but neither instruction appears anywhere in the most recent publicly available revision of the AMD64 Architecture Programmer's Manual [38], and the kernel's own definition of the SVM intercept vectors carries no entry for either of them [45], indicating that the architectural asymmetry documented in 2016 persists on current AMD platforms.

The absence of a dedicated intercept bit does not, however, constitute a formal guarantee that RDRAND is unreachable through alternative means. AMD SVM supports a small number of other mechanisms through which a hypervisor can influence instruction execution. Full instruction emulation, described in Section 4.2.2, is theoretically applicable but requires AMD SVM to be configured in a mode that emulates specific instructions in software, and AMD does not provide a general "intercept all instructions" mode that would make this practical without a dedicated bit. The Decode Assist mechanism [38, Sec. 15.10] provides instruction decode information alongside certain VMEXITs triggered by exceptions, but it is not an independently configurable interception path and cannot be used to create a VMEXIT on an instruction that would otherwise execute natively. Binary rewriting at the kernel level, also discussed in Section 4.2.2, could substitute a controlled sequence for RDRAND at translation time, but this approach is not implemented in standard KVM and would require substantially more invasive modification than the Intel proof-of-concept.

The practical consequence is that the AMD behavior is established from the specification and the source rather than from a test. The kernel’s SVM intercept definitions provide no constant for RDRAND or RDSEED [45], so there is no call to `svm_set_intercept` the KVM SVM path could make for either instruction, and the exit code that such an interception would have to report does not exist either. The software layer therefore makes no attempt to configure interception, and there is no documented hardware path it could configure. The conclusion the BSI 2016 study drew from the manual holds on current AMD platforms for the same reason it held then, which is that the mechanism the Intel attack depends on has no counterpart in SVM.

4.4 VMM Impact on the Linux Entropy Pool

Sections 4.2 and 4.3 described the mechanisms through which a virtual machine monitor can observe and alter guest execution, together with their application to the CPU hardware random number instructions in particular. The present section moves from mechanism to consequence and considers what these capabilities imply for the Linux entropy pool as a whole. Section 4.4.1 assesses each of the entropy sources enumerated in Section 3.2.3 under virtualized conditions. The boot phase, where the consequences are most acute, is examined in Section 4.4.2, which also compares the current situation against the findings of the BSI 2016 study. Section 4.4.3 then returns to the combiner guarantee established in Section 3.2.2 and determines under which conditions it continues to hold once a VMM sits between the guest and the hardware.

Unless stated otherwise, the assessment that follows assumes a hypervisor that is not actively malicious, the same assumption under which the BSI 2016 study operated [3, Sec. 2.1]. On that basis, any degradation of an entropy source follows structurally from virtualization and does not indicate an attack. Points at which the adversarial case diverges substantially are noted as they arise, while the systematic treatment of a malicious hypervisor follows in Section 4.4.3 and in Chapter 5.

4.4.1 Entropy Sources in Virtualized Environments

Section 3.2.3 enumerated the entropy sources contributing to the input pool of the current Linux kernel and described what each of them collects. The same set of sources is evaluated here under the conditions of a cloud virtual machine, along three criteria. Availability concerns whether the source exists in a virtualized guest at all. Quality concerns how much usable entropy it still delivers relative to a physical system. Mediation concerns whether the events the source observes reach the guest through the hypervisor, which governs whether a source can remain trustworthy in an adversarial deployment even when it is available and productive. Virtualization introduces the third criterion, and it cuts across the first two. A source may be present and statistically healthy and still fail as a trust anchor if the party it depends on is also the party the guest may need to distrust.

Interrupt timing (`add_interrupt_randomness`) remains the most productive software source in a guest, though it is

affected on two fronts. The interrupts a guest observes do not originate from physical devices operating independently of the virtualization layer. They are delivered by the hypervisor, either as emulated device events or as forwarded physical interrupts. Such delivery smooths the fine-grained arrival jitter from which the source draws its entropy, since the hypervisor batches, defers, and schedules interrupt injection according to its own logic and not according to physical device behavior. The entropy contribution therefore falls below what the same source yields on bare metal. The reduction in quality is only one aspect. The hypervisor also knows the delivery times it has chosen, so the values the guest treats as unpredictable remain reconstructable for it.

Disk I/O timing (`add_disk_randomness`) has been reduced by two independent developments. The shift from rotating disks to solid-state storage removed the mechanical seek variability that made this source productive when the BSI 2016 study was written, and this reduction applies on physical systems as well. Virtualization compounds the effect, because guests access storage through virtual block devices whose completion timing is determined by the hypervisor’s I/O path and not by the physical medium. A source that the BSI 2016 study still treated as one of three principal noise sources contributes only marginally in a contemporary cloud guest.

HID event timing (`add_input_randomness`) is effectively absent. Headless servers and cloud instances have no attached keyboards or pointing devices, so the source produces no events at all. One of the three noise sources on which the BSI 2016 analysis was built is therefore unavailable in the deployment context that motivates this paper [24, Sec. 3.5.2.7].

Device identifier injection (`add_device_randomness`) remains available and is unaffected in the sense that matters for it, given that it contributes no entropy credit in the first place. Its function in a cloud context nevertheless deserves mention. By mixing MAC addresses, serial numbers, and comparable identifiers into the pool, it ensures that virtual machines instantiated from an identical image diverge as soon as their distinct identifiers are absorbed. This addresses a failure mode specific to cloud deployment, where large numbers of guests are created from the same template.

The hardware RNG framework (`add_hwgenerator_randomness`) is available in a guest whenever the hypervisor exposes a `virtio-rng` device [24, Sec. 3.9.2], and where it is present it supplies entropy at a useful rate. Its dependence on the host is nevertheless total, since the entropy originates there and is passed to the guest by the hypervisor. A trusted hypervisor makes this an effective source, and one that cloud operators can and do provide deliberately. Against an adversarial hypervisor it offers nothing, because the values it delivers are chosen by the party the guest is trying to defend against.

RDRAND and RDSEED occupy a different position from every other source in this list. They execute within the CPU and depend on no peripheral events, on no device emulation, and on no interrupt delivery, so their operation is structurally independent of the mediation that weakens the software

sources. Their independence accounts for their role as the natural candidate to compensate for the entropy deficit of virtual machines, as set out in Section 3.2.4. That independence has limits, since the interception mechanisms examined in Section 4.2 apply to these instructions, and Section 4.3 examines what a VMM can do with them in practice. Degradation and interception must therefore be kept apart. The software sources are weakened by virtualization as a structural side effect, whereas RDRAND remains structurally unaffected and is vulnerable only to deliberate intervention.

Bootloader entropy (`add_bootloader_randomness`) is available in guests booted through UEFI firmware and contributes at the earliest point in the boot sequence, where it is most valuable. The entropy it supplies is produced by the virtual firmware, however, and ultimately derives from the host. It therefore shares the dependence of `virtio-rng`, being useful under a cooperative hypervisor and worthless against a hostile one.

VM fork detection (`add_vmfork_randomness`) addresses a failure mode that arises only under virtualization and is therefore more relevant in a guest than anywhere else. Its operation depends on the VM Generation ID, an identifier the hypervisor maintains and updates when a guest is cloned or restored from a snapshot. The mechanism provides genuine protection and closes a gap that the BSI 2016 study did not yet address. It remains conditional on hypervisor cooperation, because a hypervisor that fails to update the generation identifier, whether through misconfiguration or deliberately, leaves the guest unaware that its state has been duplicated.

The auxiliary sources contribute little in this context. Scheduler-based collection remains available, but the scheduling events it observes occur within a guest whose CPU time is itself allocated by the hypervisor, so the same mediation argument applies. Power management events are marginal on server workloads regardless of virtualization. The Latent Entropy GCC plugin operates at compile time and is therefore

unaffected by virtualization, though for the same reason it contributes nothing to the entropy available at runtime.

The following table summarizes the assessment. Row order follows the enumeration in Section 3.2.3 so that the two tables can be read against one another.

The assessment shows a consistent pattern. Of the sources contributing to the input pool, one is absent in a cloud guest, several are measurably weakened by the virtualization layer, and most of those that remain productive depend on the hypervisor for the events or values they collect. RDRAND and RDSEED form the exception, and they do so because they bypass every layer at which the hypervisor mediates. Their vulnerability lies elsewhere, in the interception mechanisms of Section 4.2 and not in structural degradation.

Measured against the BSI 2016 baseline, the change has two directions. Of the three noise sources that study analyzed, one has disappeared from the deployment context considered here, one has been substantially reduced by developments partly unrelated to virtualization, and one remains but delivers less than it did on the physical systems of that era. The set of sources has at the same time grown, and the additions are directed at problems the 2016 analysis identified or did not yet face, among them the boot-time gap and the duplication of guest state. Most of these additions are hypervisor-dependent, so the expansion of the source list does not translate into a proportional expansion of independent trust anchors. Section 4.4.2 examines what this concentration means at the point where entropy is scarcest, namely the boot phase.

4.4.2 Boot-Time Entropy Starvation

The assessment in Section 4.4.1 described the entropy sources of a running guest. During the boot phase the situation is more constrained still, for two reasons that reinforce one another. The software sources have not yet accumulated meaningful entropy, since interrupt and disk events require time to occur, so the already reduced set of sources described above shrinks

TABLE II
THE SAME SOURCES SEEN FROM INSIDE A VIRTUAL MACHINE.

Source	Availability in VM	Quality impact	Hypervisor-mediated
<code>add_interrupt_randomness</code>	Available	Reduced (jitter smoothed by delivery)	Yes
<code>add_disk_randomness</code>	Available	Strongly reduced (SSD plus virtual block layer)	Yes
<code>add_input_randomness</code>	Absent	No contribution	Not applicable
<code>add_device_randomness</code>	Available	Unchanged (no entropy credit)	Partly (identifiers assigned by host)
<code>add_hwgenerator_randomness</code>	Conditional (<code>virtio-rng</code> exposed)	Good where present	Yes (entropy originates on host)
<code>add_bootloader_randomness</code>	Available (UEFI guests)	Good	Yes (supplied by virtual firmware)
<code>add_vmfork_randomness</code>	Available	Not an entropy contribution, triggers reseed	Yes (depends on VM Generation ID)
Scheduler-based	Available	Reduced	Yes (scheduling governed by hypervisor)
Power management	Available	Marginal on servers	Partly
Latent Entropy GCC Plugin	Available	Static, no runtime contribution	No
RDRAND / RDSEED	Available	Unaffected structurally	No (subject to interception, see 4.3)

further at exactly the moment the system starts. The boot phase is at the same time when a system generates the credentials that persist longest.

The concern applies most acutely to the first boot of a newly created guest. SSH host keys are generated when the daemon starts for the first time and typically remain in service for the lifetime of the machine, so a deficit at that point propagates for years. Subsequent reboots are less exposed, since the long-lived material already exists on disk and most distributions save a seed at shutdown and restore it during the following boot, leaving the pool with a non-empty starting state [24]. That mechanism carries its own difficulty under virtualization. A cloned image contains the seed of the template it was derived from, so every guest instantiated from that image restores the same value, one of the failure modes that `add_device_randomness` and `add_vmfork_randomness` are intended to address.

Section 2.1.2 established the formal basis for treating this as a cryptographic weakness, not merely a statistical curiosity. Conditional min-entropy accounts for what an adversary knows in addition to the output itself, and knowledge of a machine's start time constrains the set of plausible pool states considerably. An adversary who can observe or estimate when a guest was launched knows that only a limited number of events can have contributed to the pool, which reduces the effective unpredictability well below what the unconditional measure suggests.

The BSI 2016 study quantified the problem on the systems of its time. On a KVM-based guest, the Linux entropy pool required between 25 and 90 seconds to accumulate 128 bits of entropy after boot, while the operating system itself completed initialization within roughly two seconds [3]. Cryptographic daemons starting in that interval drew on a pool that had not yet reached the threshold the kernel treated as sufficient. The 128-bit figure belongs to the entropy accounting of the architecture analysed then, whereas the current kernel uses a threshold of 256 bits under a different accounting model, so the two numbers describe comparable concerns without being directly comparable quantities.

Three changes since then bear on this window. Bootloader entropy delivered through the UEFI random number protocol contributes before the kernel has collected any events of its own [24, Sec. 3.5.2.6]. The replacement of continuous entropy accounting with a single initial seeding threshold removed the recurring blocking behavior of the legacy design. The most consequential change is that the kernel queries the CPU hardware entropy source directly during early boot, preferring RDSEED and falling back to RDRAND where RDSEED is unavailable. Where that source is credited, it supplies the full 256-bit threshold in a single sequence of instructions, so the pool reaches the fully seeded state within a fraction of a second [24, Sec. 3.3.1.3]. The interval that the BSI 2016 study measured in tens of seconds effectively disappears.

The measurements taken for this paper confirm that picture with figures of their own. On bare metal on the AMD EPYC

9354, the pool reached the fully seeded state after 0.1327 s with the CPU source credited and after 7.9360 s without it, each figure the mean of three boots, and a second pair of configurations differing in whether bootloader entropy was credited gave 0.1332 s and 8.1164 s. The cost of withholding the credit is therefore a difference of about two orders of magnitude in the length of the exposed window rather than a difference in kind, and crediting the bootloader changes nothing that is visible at this resolution. Section 7.5 reports the measurement in full, including what draws on the pool during the longer interval. The figures come from the host platform. No corresponding measurement was taken inside a confidential guest.

This improvement warrants a precise characterisation. The software sources have not become faster or more productive under virtualization, and Section 4.4.1 indicates that several of them have become less so. The change lies in the addition of a source that renders the accumulation time of the others irrelevant for the purpose of initial seeding. Substitution closes the boot-time gap, not repair.

The blocking behavior of `getrandom()` and `/dev/random` remains the mechanism that formally protects against premature key generation, since both wait until the seeding threshold has been reached. Under the default configuration this protection rarely engages, because the CPU source satisfies the threshold before any cryptographic daemon issues a request. Blocking therefore functions as a fallback safeguard for the cases where the fast source is absent or distrusted. Two categories of consumer fall outside its protection in any case. Software that reads `/dev/urandom` directly is served without waiting, and kernel-internal consumers that require randomness before the pool is seeded cannot defer their requests at all. Kernel address space layout randomisation is the clearest instance, since the early boot code must compute the physical and virtual KASLR offsets before the kernel image is decompressed, long before any entropy pool exists, and draws them instead from RDRAND and the timestamp counter directly [46], [44]. The stack canary of the boot CPU is established under the same constraint and was for a period initialised from the timestamp counter alone, precisely because no pool was available at that point in the boot sequence [47]. Both mechanisms therefore depend on the CPU instruction at a moment when no alternative source has yet accumulated anything, which is the same concentration that Section 4.4.3 examines in general terms.

The configuration in which the CPU source is not credited returns the guest to the position Section 4.4.1 described. With `random.trust_cpu` set to N, RDSEED output continues to be mixed into the pool but no longer advances the entropy estimate [24, Sec. 3.3.1.4], so the seeding threshold must be reached through the software sources alone. In a cloud guest these are precisely the sources that virtualization has weakened, and one of the three that the BSI 2016 study relied upon is absent entirely. The system does not thereby fail to run, since the kernel itself continues to operate. Stalling occurs instead among the consumers waiting on `getrandom()` or `/dev/random`, so services that require randomness during startup do not proceed until the threshold is reached, and in a

guest with little device activity that wait can be considerable. How much of the system draws on the pool before it is seeded is visible once the kernel is asked to report it. A boot of the AMD platform with `trust_cpu` set to 0 recorded 59,938 accesses to the unseeded pool, spread over 32 distinct callers, before the threshold was reached. A single caller accounts for most of them, since 55,248 of the accesses come from `__get_random_u32_below`, while the remainder are spread over process creation, address space randomisation, filesystem and key management. That boot ran a 6.18 kernel built with the option that reports every such access, and it carried an enlarged log buffer in order to capture the reports at all, which moved the seeding time to 9.175 s, so it supports the list of consumers and not the timing figures, which are taken from the boots described above. A hypervisor providing `virtio-rng` offers a way out, because entropy then reaches the pool through the hardware RNG framework without depending on the CPU instruction. That path relocates the trust assumption to the cloud operator without removing it, which leaves it an unsatisfying resolution for an operator who set `trust_cpu` to N out of distrust toward the CPU vendor in the first place. Distrusting the CPU source therefore exchanges a trust risk for an availability risk, and the severity of the exchange is greater in a virtualized guest than on the physical systems of 2016. This trade-off explains why `trust_cpu` defaults to Y across current distributions.

The configuration is in any case a statement about what the guest is willing to credit, and it provides no defense against interference. A hypervisor that intercepts the instruction determines the value the guest receives regardless of how the guest has configured `trust_cpu`. With the source credited, the guest treats an injected value as fully entropic and considers itself securely seeded. With the source uncredited, the injected value is still mixed in, and the guest instead depends on software sources that the same hypervisor also mediates. No setting available to the guest changes what a hypervisor in that position can do, which is why the discussion of confidential computing in Chapter 5 concerns architecture and not configuration.

On this evidence, the boot-time entropy problem identified in 2016 has been displaced more than resolved. In 2016 the question was whether a system could accumulate sufficient entropy before it needed it. The question today is whether the source that fills the threshold immediately deserves the credit it is given. Under the current architecture, the trustworthiness of `RDRAND` and `RDSEED` is not an adjacent consideration to the boot problem. It is the form the boot problem now takes.

One failure mode remains outside the reach of the blocking mechanism entirely. A guest restored from a snapshot inherits a pool that the kernel considers fully seeded, so nothing blocks and no threshold is re-evaluated, yet every guest derived from that snapshot begins from an identical state. The kernel addresses this through `add_vmfork_randomness`, which triggers reseeded when the VM Generation ID changes [24, Sec. 3.5.2.7], and Section 4.4.1 noted that the mechanism depends on the hypervisor maintaining that identifier correctly. Section 4.4.3 takes up the question whether the combiner can

compensate for a situation in which the trustworthy contributions have narrowed to a single source.

4.4.3 The Combiner under VMM Interference

Section 2.2.3 established the guarantee on which the Linux combiner rests. Where several sources are combined cryptographically, the result retains at least the min-entropy of the strongest contribution, provided that contribution is unknown to the adversary. The guarantee itself is not at issue here. At issue is whether its premise survives virtualization, and the assessment in Section 4.4.1 supplies the material for answering that.

The premise has two parts, and they fail in different ways. A source may contribute too little entropy to be useful, which is a question of quality, or it may contribute entropy the adversary already knows, which is a question of exposure. The combiner accommodates the first case by construction, since a weak contribution neither helps nor harms and the guarantee attaches to whichever source is strongest. The second case lies outside what the combiner can address. A source whose values the adversary can reconstruct contributes nothing to unpredictability from the adversary's position, however healthy it appears from within the guest. This distinction matters because virtualization affects the two differently. Section 4.4.1 documented reductions in quality across the software sources, and those reductions are the less consequential finding. The decisive column is the one recording whether each source is mediated by the hypervisor.

Read against the premise, that column describes a systematic problem. Interrupt timing is exposed, since the hypervisor selects the delivery times from which the entropy is drawn. Values obtained through `virtio-rng` are produced by the host and are exposed for that reason. Bootloader entropy originates in virtual firmware under hypervisor control. Scheduling events occur within CPU time the hypervisor allocates. Each of these sources continues to deliver values that satisfy every statistical test the guest could apply to them, and each is known to the party against which the guest would need protection. Only `RDRAND` and `RDSEED` escape the pattern, and they escape it by depending on no mediated event, as Section 4.4.1 set out.

The guarantee therefore narrows to a single contribution. Where the hypervisor is not adversarial, this narrowing has no practical consequence, since the mediated sources are merely weakened and remain unknown to any party the guest is defending against. Where the hypervisor is adversarial, the guarantee rests entirely on whether the CPU instructions remain outside its reach, and Section 4.3 examined the mechanisms through which they may not.

The combiner thus carries an assumption it cannot verify. Combining multiple sources protects against the failure of any one of them on the understanding that their failure modes are independent, so that a defect in one implies nothing about the others. Virtualization violates that understanding without attacking the construction. The sources feeding a guest's entropy pool do not fail independently, because most of them

depend on the same actor for the events they observe, so a single adversarial position compromises many of them at once. Reliability engineering describes this as a common-mode failure, and a design whose security argument rests on independence offers no defense against one [48]. In such a scenario the combiner has not been circumvented at all, since the environment has removed its premise.

Nothing in the mechanism signals when this occurs. The combiner performs no assessment of the inputs it absorbs, so it cannot distinguish a source contributing genuine unpredictability from one whose values are chosen elsewhere, and it continues to produce output that no statistical examination would find wanting. A guest has no way of observing the transition from a state in which the guarantee holds to one in which it does not.

The analysis leaves the security of the entropy pool in a conventional cloud guest dependent on a single question, namely whether RDRAND and RDSEED remain outside the hypervisor’s control. The combiner cannot compensate for that concentration, since compensation would require an independent source, and virtualization has removed the independence. Chapter 5 takes up the question whether an architecture can restore it by placing at least one source beyond the reach of the hypervisor instead of trusting that one remains untouched.

5 CONFIDENTIAL COMPUTING AS A TRUST MODEL SHIFT

5.1 Eliminating the Trusted VMM Assumption

The BSI 2016 study operated under an explicit assumption that the hypervisor is not actively malicious, and that assumption was a reasonable boundary for the question it was asking. For the deployment contexts that motivate this paper, the assumption does not hold. A cloud provider is an external party whose interests may diverge from those of the tenant. A hypervisor that has been compromised by a state-level actor, modified by an insider, or subjected to a supply-chain attack presents exactly the threat that the conventional architecture cannot address by design. Compliance frameworks operating at BSI high-security levels require that trust assumptions be explicitly justified rather than postulated, and an architecture in which the security of the entropy pool depends on the hypervisor behaving honestly does not meet that requirement.

The question this chapter addresses is whether the trusted VMM assumption can be replaced by an architectural guarantee rather than an act of trust. Confidential computing answers this question affirmatively by redesigning where the isolation boundary is drawn. Instead of placing the hypervisor inside the Trusted Computing Base (TCB) and relying on it to behave correctly, confidential computing hardware enforces isolation at the CPU level so that the hypervisor is excluded from the TCB by construction. The hypervisor continues to manage scheduling and resource allocation, since those functions require no access to guest memory or CPU state, but

it is architecturally prevented from reading or modifying the contents of a protected virtual machine. The trust relationship changes from one that must be extended voluntarily to one that the hardware enforces regardless of the hypervisor’s behavior or intentions.

The consequence for entropy is direct. If the hypervisor cannot access the execution context of a confidential guest, it cannot configure VM-exits for instructions executing within that context. The interception attack demonstrated in Section 4.3.1 depends on the ability to modify the VM-execution controls, which are part of the state the hypervisor manages. Within a hardware-enforced isolation boundary, that ability is removed. RDRAND becomes not merely structurally independent of hypervisor mediation but architecturally protected against the deliberate interference that Section 4.3 showed was otherwise possible. What is removed is the instruction intercept, and not every path by which the host can shape what a guest does with the instruction. Under SEV-SNP the hypervisor still supplies the CPUID values a guest sees, subject to the firmware policy that Section 5.6 examines, so it retains an indirect route to the question of whether a guest uses the instruction at all. What this implies for the entropy pool as a whole, and for the sources that remain outside this protection, is the subject of Section 5.5.

This shift does not resolve every concern about RDRAND. The vendor opacity problem identified in Section 3.3.3 is not addressed by confidential computing, since TDX and SEV-SNP provide no independent window into Intel’s or AMD’s entropy source implementation. The trust requirement narrows but does not disappear. Where a conventional cloud deployment requires trusting both the CPU vendor and the hypervisor operator, a confidential computing deployment narrows that to the CPU vendor, because the hypervisor operator is no longer in a position to intercept the instruction itself, although the paths that Section 5.6 collects leave it some influence over the conditions under which the guest runs. Whether that residual trust is sufficient is a question the empirical analysis in Chapter 7 and the discussion in Chapter 8 address. This chapter establishes the architectural foundations needed to evaluate that question, covering Intel TDX and AMD SEV-SNP as the two dominant implementations of hardware-enforced confidential computing on server platforms, remote attestation as the mechanism through which a guest can verify its own isolation, and the reassessment of entropy sources that follows once the hypervisor is removed from the trust boundary.

5.2 Intel TDX

5.2.1 Architecture and Trusted Computing Base

Intel Trust Domain Extensions (TDX) introduce a new CPU privilege level that sits above VMX root mode, the level at which the hypervisor normally operates, and thereby create a protected execution environment that the hypervisor cannot read or modify. Understanding how this works requires following the architecture from the hardware up, because the

isolation guarantee is a property of the CPU itself rather than of any software component above it.

The central architectural innovation is Secure Arbitration Mode, or SEAM. SEAM is a new CPU operating mode defined by an isolated memory range whose boundaries are fixed by the SEAM Range Registers during platform initialization. Code executing in SEAM cannot be observed or interrupted by code running outside it, including the hypervisor operating in VMX root mode. This isolated range is where the TDX Module lives, which is an Intel-signed software module that acts as a secure intermediary between the hypervisor and the Trust Domains it manages. The hypervisor communicates with the TDX Module exclusively through the SEAMCALL instruction, which transfers control into the SEAM range in a controlled and auditable way. The TDX Module itself enforces the security policies that protect Trust Domains and cannot be replaced or modified by any software component above it, because its cryptographic signature is verified by the CPU before it is permitted to load into the SEAM range.

Memory isolation is enforced through Multi-Key Total Memory Encryption, or MKTME [49]. Every isolated execution environment, also called Trust Domain or TD for short, receives a unique encryption key that the CPU uses to transparently encrypt and decrypt all memory belonging to that TD as it moves between the processor and RAM. The keys are managed as TDX-private KeyIDs that are accessible only within SEAM mode, meaning the hypervisor, which operates outside SEAM, never has access to the keying material. When the hypervisor reads from the physical addresses that back a TD's memory, it receives only ciphertext. The practical consequence is that memory isolation does not depend on the hypervisor voluntarily refraining from reading guest memory but rather on physics and cryptography, which the hypervisor cannot override.

The CPU register state of a running TD is protected by the same boundary. When a TD is preempted or exits for any reason, the TDX Module saves and encrypts the TD's register state before returning control to the hypervisor. The hypervisor never sees the plaintext register contents. This property is directly relevant to the RDRAND interception mechanism described in Section 4.3.2, which relied on the hypervisor's ability to read and write the VMCS state-save area containing the guest's register file. Within a Trust Domain, the TDX Module holds exclusive authority over the TD's execution controls and state-save areas, and the hypervisor has no write path into them. The configuration of VM-execution controls, including the RDRAND exiting and RDSEED exiting bits examined in Section 4.3.2, falls under the TDX Module's authority rather than the hypervisor's, which is the mechanism through which TDX eliminates the interception attack at the architectural level rather than relying on the hypervisor to refrain from mounting it. The specification supports this reading by omission, since the instructions a Trust Domain may not execute or that are reflected to it are enumerated in full and include neither RDRAND nor RDSEED [5, Sec. 11.7], and the published module source settles it directly, because it initialises both exiting controls to zero and keeps

them out of the mask of controls a host is allowed to vary [6]. This is an architectural finding and not a measured one. No Trust Domain could be created on the Intel platform available for this work, for the reasons Section 6.2.4 records, so the claim rests on the specification and the source rather than on an experiment.

The Trusted Computing Base under TDX consists of three components. The Intel CPU hardware itself, the TDX Module running in the SEAM range, and the software stack executing inside the Trust Domain. The hypervisor lies outside it, which does not mean that it is absent. It continues to manage scheduling, memory allocation, and I/O, but its correct behavior is not a prerequisite for the confidentiality and integrity of what runs inside a TD. A hypervisor that is compromised, misconfigured, or operated by an adversarial party cannot read TD memory, cannot observe TD register state, and cannot manipulate the execution controls of a running TD. TDX version 1.0, available from the 4th generation Intel Xeon Scalable processors (Sapphire Rapids) onward and more widely deployed in the 5th generation (Emerald Rapids), provides this guarantee for the software and memory of the TD. The TDX Module Base Architecture Specification was most recently updated in April 2025 and defines the interface and security properties of the module that all compliant implementations must satisfy [49].

5.2.2 What TDX Does and Does Not Protect

The isolation guarantees described in Section 5.2.1 are precise and should not be overstated. TDX provides strong confidentiality and integrity for the memory and CPU state of a Trust Domain against a malicious hypervisor, and it eliminates the VM-exit based interception attack on RDRAND that Section 4.3 examined. Several other threat vectors remain unaddressed, and understanding their scope is necessary for correctly interpreting the entropy analysis that follows in Section 5.5.

The Linux TDX guest kernel security specification states explicitly that interrupt timing within a TD is controlled by the untrusted VMM, and that this makes interrupt timing unsuitable as an entropy source inside a TDX guest. The hypervisor schedules and delivers interrupts to the TD, so the timing values that the Linux entropy pool would normally collect from interrupt arrival are values the hypervisor chooses. This is not a consequence of any gap in TDX's memory or register protection but a structural property of how interrupt delivery works in the x86 architecture, and TDX does not alter it. The practical consequence is that the `add_interrupt_randomness` source, which Section 4.4.1 identified as the most productive software source in a conventional guest, cannot be trusted inside a TDX guest for the same reason it could not be trusted in a conventional guest with a malicious hypervisor because the values it observes are determined by the untrusted party. This is the reason the Linux kernel commit requiring RDRAND seeding on confidential computing systems was introduced, and why the Linux TDX guest kernel security specification mandates that 256 bits of RDRAND

output be mixed into the entropy pool during early boot before the boot process is permitted to continue.

I/O remains under hypervisor control in TDX 1.0. Virtual network interfaces, storage controllers, and other emulated devices are managed by the hypervisor, which means virtio-rng, bootloader entropy supplied through virtual firmware, and any other entropy that reaches the guest through an emulated device path originates outside the TD's trust boundary. TDX 2.0 introduces TEE-I/O, which extends the isolation boundary to cover directly attached devices, but this extension is not yet widely deployed and falls outside the scope of this paper. The entropy assessment in Section 5.5 treats I/O-dependent sources as hypervisor-mediated and therefore untrusted inside a TDX guest.

It should be pointed out that availability is not protected. The hypervisor retains the ability to pause, resource-starve, or terminate a Trust Domain. TDX's security model is concerned with confidentiality and integrity of what runs inside a TD, not with guaranteeing that it runs at all. This is a deliberate design boundary and does not affect the entropy analysis, but it is relevant context for deployments that require availability guarantees alongside confidentiality ones.

Microarchitectural side-channel attacks represent a residual risk that TDX mitigates but does not eliminate. A security review of TDX carried out by Google Project Zero together with Google Cloud Security on the pre-release source code of TDX version 1.0, published in April 2023, evaluated 81 attack scenarios and identified ten confirmed security vulnerabilities, all of which were fixed before the products shipped, alongside five further areas recommended for defence-in-depth hardening in later versions [50]. The exercise demonstrates that the microarchitectural attack surface against TDX is real and has required active remediation, and that future generations may surface additional issues as the technology matures and receives broader scrutiny.

The vendor trust residual is the most consequential limitation for the analysis in this paper. The TDX Module is Intel-signed and its integrity is verified by the CPU before it loads into the SEAM range, but both the module and the hardware that enforces its authority are Intel products. The trust reduction introduced in Section 5.1 therefore leaves Intel inside the TCB rather than removing it. The same opacity that prevents independent auditing of RDRAND's physical entropy source also applies to the TDX Module. Its source code is not publicly available, and trust in its correctness rests on Intel's assurance and on the security reviews that external parties have been permitted to conduct. This residual trust assumption is returned to in Section 5.6 and in the discussion of Chapter 8.

5.3 AMD SEV-SNP

5.3.1 Architecture and Comparison to TDX

AMD's approach to confidential computing has evolved through three successive generations of the Secure Encrypted Virtualization family. The first generation, SEV, introduced

per-VM memory encryption so that the hypervisor would see only ciphertext when accessing guest memory. The second generation, SEV-ES, extended this protection to the CPU register state by encrypting register contents during world switches, preventing the hypervisor from reading or writing guest registers at preemption boundaries. The third and current generation, SEV-SNP, added memory integrity guarantees through a new hardware mechanism called the Reverse Map Table, closing the attack vector through which a hypervisor could remap encrypted pages to different physical addresses while leaving the encryption intact. Only SEV-SNP provides the full set of confidentiality, integrity, and isolation guarantees that make it relevant to the analysis in this paper.

The hardware root of trust in SEV-SNP is the Platform Security Processor, or PSP, an ARM-based coprocessor that is integrated directly onto the CPU die but operates entirely independently of the main x86 cores. Unlike Intel's TDX Module, which is a loadable software module that resides in the SEAM memory range and can be updated independently of the CPU firmware, the PSP is a dedicated hardware component whose firmware is part of the CPU's platform firmware stack. The PSP manages all encryption keys for SEV-SNP VMs, initializes the Reverse Map Table, and is the sole component authorized to generate and sign attestation reports. Because the PSP operates independently of the x86 cores and the hypervisor has no execution path into it, the PSP serves as the attestable anchor of the trust chain in the same role that the TDX Module plays in the Intel architecture, though through a different implementation philosophy.

Memory encryption operates on the same principle as in TDX. Each SEV-SNP VM receives a unique encryption key managed exclusively by the PSP, and the on-die memory controller transparently encrypts all data belonging to the VM as it crosses the die boundary. The hypervisor, operating on the x86 cores outside this boundary, observes only ciphertext when accessing guest private pages. What SEV-SNP adds over its predecessors is the Reverse Map Table (RMP), which enforces integrity on top of this confidentiality guarantee. The RMP is a system-wide data structure that maintains one 16-byte entry for every 4 KB page of physical memory. Each entry records the ownership of the page, the expected guest physical address to which it should be mapped, and the access permissions granted at each privilege level. When any memory access occurs, the CPU memory controller performs an inline RMP check to verify that the accessing entity is the declared owner and that the translation is consistent with the recorded state. A hypervisor that attempts to remap a guest private page to a different physical address, or to map the same physical page into multiple guest address spaces simultaneously, will have the access blocked by the RMP check before it reaches the encrypted data. This hardware-enforced ownership tracking is the mechanism that closes the page remapping attacks that affected earlier SEV generations.

The consequence for register state protection follows directly from the combination of memory encryption and SEV-ES register state encryption. When a SEV-SNP VM is preempted, the CPU saves the guest register state into a structure called

the VM Save Area, or VMSA, which is itself encrypted with the VM's private key and marked as owner-exclusive in the RMP. The hypervisor cannot read the VMSA in plaintext and cannot modify its contents. This is the architectural mechanism that prevents the RDRAND interception attack described in Section 4.3.2. The guest's register file, including the target register that RDRAND would write its output into, is inaccessible to the hypervisor both during execution and at preemption boundaries. There is no equivalent to the VMCS state-save write path that the BSI 2016 proof-of-concept exploited, because the VMSA is in encrypted guest-owned memory that the hypervisor cannot reach.

Communication between the guest and the hypervisor uses a different mechanism than TDX's SEAMCALL interface. SEV-SNP introduces a new exception class, #VC, which the CPU raises when the guest executes an instruction that requires hypervisor involvement. The guest kernel handles the #VC exception and decides what information to share with the hypervisor before requesting assistance. This pull model, where the guest controls what it discloses, contrasts with a model in which the hypervisor can freely query guest state, and it is an explicit part of the SEV-SNP design to keep the guest in control of the information boundary.

SEV-SNP also introduces VM Privilege Levels, numbered VMPL 0 through VMPL 3, which allow privilege separation within an encrypted guest. VMPL 0 is the most privileged level and is used in some deployment models for a thin firmware layer that handles #VC exceptions and attestation on behalf of the less privileged guest operating system running at VMPL 1. This design allows a platform-provided component to assist with host communication without requiring the guest OS itself to implement the full SEV-SNP guest protocol. Azure's confidential VM implementation uses this model through a thin firmware layer called the Paravisor that runs at VMPL 0 and handles attestation and host communication on behalf of the guest operating system running at VMPL 1, demonstrating how the privilege separation within a SEV-SNP VM can be used to isolate trust-sensitive operations from the general guest workload.

The two architectures pursue the same security goal through distinct engineering philosophies. TDX introduces a new CPU execution mode above the hypervisor and places a loadable Intel-signed module there to mediate all interactions between the hypervisor and Trust Domains. SEV-SNP keeps the hypervisor at its existing privilege level but constrains what it can do through hardware-enforced data structures, specifically the RMP, that run in parallel with the normal memory access mechanisms. The TDX approach is more centralized, with the TDX Module as a single enforcement point, while the SEV-SNP approach is more distributed, with enforcement embedded in the memory controller itself. Both approaches exclude the hypervisor from the TCB with respect to guest confidentiality and integrity, and both leave the hypervisor in control of scheduling, I/O, and resource allocation.

5.3.2 What SEV-SNP Does and Does Not Protect

The protection guarantees of SEV-SNP follow the same general shape as those of TDX, with guest memory confidentiality and integrity, CPU register state protection, and remote attestation as the core capabilities, and with availability, I/O, and microarchitectural side channels as explicit exclusions from the threat model. Several properties of the SEV-SNP design and its recent vulnerability history deserve specific attention in the context of this paper.

The entropy situation inside a SEV-SNP guest mirrors the TDX case described in Section 5.2.2, meaning interrupt timing is controlled by the untrusted hypervisor and cannot be treated as an independent entropy source, and the Linux kernel enforces the same RDRAND boot-time requirement here as well. The full cross-platform assessment follows in Section 5.5. The two architectures are treated symmetrically in that analysis, but not in the measurements, since guest measurements were obtained on the SEV-SNP side alone and the TDX side rests on the specification and the source, as Chapter 6 records.

A vulnerability discovered in 2025 and published at ACM CCS under the name RMPocalypse is directly relevant to the trust analysis in this paper. The attack exploits a structural weakness in the RMP initialization sequence. Because the RMP itself must be stored in DRAM and the RMP's own entries are used to protect the RMP from modification, there is a bootstrapping problem during platform initialization. The RMP cannot protect itself before it has been set up, and setting it up requires that it be writable. AMD's solution is to delegate RMP initialization to the PSP, which holds exclusive write access to the RMP region during the initialization window while x86 core accesses are fenced. The researchers showed that this fencing is incomplete, allowing a malicious hypervisor to perform a single targeted write to the RMP during the initialization phase, which is sufficient to corrupt the integrity guarantees for all subsequently created SEV-SNP VMs on that platform. AMD acknowledged the vulnerability on 14 October 2025 as AMD-SB-3020 and CVE-2025-0033, characterising it as a race condition arising while the Secure Processor initializes the Reverse Map Table, rating it Medium under its own scheme, and released firmware updates to address it. The significance of RMPocalypse for this paper is twofold. It demonstrates that the RMP, which is the primary mechanism distinguishing SEV-SNP from its predecessors and the basis on which memory and register state are protected against modification by the hypervisor, had a concrete exploitable weakness in all deployed Zen 3, Zen 4, and Zen 5 processors at the time of writing, and it illustrates that the security of confidential computing implementations depends not only on the correctness of their architectural design but on the correctness of their initialization sequences, which are harder to verify formally and have historically been a productive attack surface. What the RMP does not do is close the interception vector, because on AMD platforms there is no RDRAND intercept to close in the first place, as Section 4.3.3 established from the manual and the kernel source and independently of SEV-SNP. Its relevance to this paper lies elsewhere, in

denying a hypervisor the ability to manipulate guest state or to place a guest in a position where an instruction is emulated on its behalf.

The vendor trust residual in SEV-SNP mirrors the situation in TDX but with a different implementation profile. Trust in SEV-SNP ultimately rests on trust in AMD as the manufacturer of the PSP and the designer of the RMP enforcement logic. The PSP firmware is updatable, which allows AMD to patch vulnerabilities such as the RMPocalypse initialization flaw, but it also means that the firmware version running on a given platform is part of the trust chain and must be verified through attestation before relying on SEV-SNP guarantees. The attestation report produced by the PSP includes a TCB version number that identifies the firmware and microcode version, allowing a remote verifier to determine whether a known-vulnerable firmware version is present and to refuse to provision secrets to platforms that have not applied available patches. This TCB versioning mechanism is one of the concrete points of comparison between SEV-SNP and TDX that Section 5.4 returns to in the context of remote attestation.

5.4 Remote Attestation of CPU Type and Enclave State

The isolation guarantees described in Sections 5.2 and 5.3 address the question of what a hypervisor cannot do to a running confidential guest. Remote attestation addresses a different but equally important question, which is how a party outside the platform can verify that those guarantees are actually in effect rather than taking the hypervisor's word for it. Without attestation, a cloud tenant who starts a virtual machine and is told by the provider that it runs inside a TDX Trust Domain or a SEV-SNP protected VM has no independent means of confirming that claim. Attestation provides that means by producing cryptographically signed evidence rooted in hardware that the hypervisor cannot forge or suppress.

An attestation report answers three questions simultaneously. That the workload is executing on genuine hardware capable of providing the claimed isolation, that the software stack inside the enclave matches what was expected, and that the firmware and microcode versions active on the platform are known and unrevoked. All three answers are bound together in a single signed artifact whose signature can be verified against a certificate chain rooted in the CPU vendor, making it impossible for the hypervisor to substitute a false report without detection.

On Intel TDX platforms, attestation is built on the Data Center Attestation Primitives infrastructure, commonly called DCAP [51]. A Trust Domain can request a TD Report through the TDCALL instruction, producing a hardware-generated structure that includes a cryptographic measurement of the TD's initial state, a TCB version number identifying the TDX Module and microcode versions currently active, and a 64-byte user-supplied field called `REPORT_DATA` that the TD can use to bind arbitrary data to the report, such as a hash of a public key or a nonce provided by a verifier. The TD Report is a local artifact that cannot be verified remotely on

its own. To make it remotely verifiable, the report is passed to a Quoting Enclave, an Intel-signed software component that transforms the TD Report into a TD Quote by signing it with an ECDSA attestation key. That attestation key is itself certified by a Provisioning Certification Key whose certificate is distributed by Intel's attestation infrastructure. A remote verifier can retrieve the relevant certificates from Intel's Provisioning Certificate Caching Service and verify the full certificate chain, confirming that the quote was produced by a genuine TDX-capable platform running the TDX Module version identified in the quote. The use of a separate Quoting Enclave that is independently updateable means that the attestation mechanism can be patched or revised without requiring CPU hardware changes.

On AMD SEV-SNP platforms, attestation is produced directly by the Platform Security Processor. A guest requests an attestation report through the sev-guest kernel module, which uses the SEV-SNP firmware ABI to send the request to the PSP. The PSP generates a report containing measurements of the guest's launch configuration, the security patch levels of the bootloader, OS, and SNP firmware, and a 64-byte `REPORT_DATA` field serving the same binding purpose as in TDX. The report is signed with the Versioned Chip Endorsement Key, or VCEK, which is derived from a chip-unique secret combined with the security version numbers of the relevant firmware components [52], meaning that a VCEK certificate is specific to both the chip and the patch level. AMD operates a Key Distribution Service from which verifiers can retrieve the certificate for any VCEK, allowing offline verification of SEV-SNP reports without a live connection to AMD's infrastructure at verification time. Cloud providers may alternatively use the Version Loaded Endorsement Key, or VLEK, which is derived from a secret shared between AMD and the provider rather than a chip-unique secret, accommodating deployment models where the provider acts as an intermediary in the trust chain.

TCB versioning is central to both attestation mechanisms and deserves particular attention in the context of this paper. Every attestation report contains an explicit record of the firmware and microcode version running on the platform at the time the report was generated. This version information allows a remote verifier to determine whether the platform has applied available security patches or is running a version known to be vulnerable. The relevance to the RMPocalypse vulnerability discussed in Section 5.3.2 is direct. A verifier that knows a given VCEK corresponds to a firmware version affected by the RMP initialization flaw can refuse to provision secrets to that platform until the patch has been applied, and the patched firmware version produces a different VCEK that the verifier can distinguish from the vulnerable one. TCB versioning thus provides the mechanism through which the confidential computing ecosystem can enforce security hygiene at scale, without requiring manual platform inspection, because the evidence of the platform's state is always bound to the attestation report that a relying party receives.

For this paper, attestation serves two concrete purposes that are worth making explicit. The first is verification of the

measurement environment. The empirical analysis in Chapter 6 compares RDRAND and RDSEED entropy quality across different environments including bare metal, a conventional KVM guest and a SEV-SNP protected VM. Without attestation, it is not possible to confirm that a cloud instance advertised as a confidential VM is actually running inside a hardware-enforced enclave rather than a conventional VM configured to look like one. Attestation is what would provide the cryptographic evidence that a measurement was taken in the claimed environment. That evidence could not be obtained for the guest measurements reported here, because the kernel image available in the rented environment carried no guest attestation driver, so the claim that those measurements come from a confidential guest rests on what the guest kernel reports about itself. Section 6.2 records the consequence, and Section 6.5 counts it among the limits of the study.

The second purpose is CPU type and firmware version identification. What a vendor has certified is product-family-specific, so that the certified entropy per RDSEED block for an EPYC 7xx2/7xx3 processor differs from the value for an EPYC 9xx4 processor, and a statement about what was measured is only intelligible once the family it was measured on is fixed. The attestation report contains the exact CPU family identification and the TCB version, and it would therefore tie a measurement taken inside a confidential guest to a particular generation and firmware state, which in the absence of a report falls back to the manifest and to what the platform reports about itself. The same identification also decides whether a known generation-specific defect applies, as Section 3.4.1 showed for CVE-2025-62626 on Zen 5.

What attestation does not provide is any evidence about entropy quality itself. A valid attestation report from an Intel TDX platform proves that the measurement was taken inside a genuine Trust Domain with the hypervisor excluded from the TCB, but it says nothing about whether RDRAND is delivering the entropy quality that Intel's ESV certificate specifies, whether the physical noise source is operating within its characterized parameters, or whether any subtle implementation weakness affects the output.

5.5 Entropy Sources under Confidential Computing

5.5.1 *Reassessment of Entropy Sources in TDX and SEV-SNP Guests*

Sections 5.2.2 and 5.3.2 established what TDX and SEV-SNP protect and where their guarantees end. This section applies those boundaries to the entropy sources of the Linux kernel, completing the third and final assessment of the same set of sources. Section 3.2.3 described what each source collects and Section 4.4.1 evaluated them under conventional virtualization. The question here is narrower than either. For each source, does confidential computing change the conclusion reached in Section 4.4.1?

The answer follows from a boundary that both architectures share. Confidential computing removes the hypervisor's access to guest memory and CPU register state, along with its

authority over the execution controls that governed the interception attack. It leaves untouched the hypervisor's role as the party that delivers interrupts, emulates devices, and allocates CPU time, since a virtual machine requires a manager and those functions demand no access to protected state. The sources of the Linux entropy pool divide cleanly along this line. Those that observe events the hypervisor supplies remain in the position Section 4.4.1 described, nothing about their situation having changed. Those that originate inside the CPU move from interceptable to protected. The division is sharper than in Section 4.4.1, where several sources occupied intermediate positions. Under confidential computing a source is either mediated by an untrusted party or it is not.

Interrupt timing remains mediated. Section 5.2.2 records that the Linux TDX guest kernel security specification classifies host-controlled randomness as a distinct threat and identifies interrupt timing as unsuitable for entropy collection inside a Trust Domain [7, Threat model and Randomness inside TDX guest]. The situation in a SEV-SNP guest is identical. This produces a tension the kernel does not resolve. The entropy accounting continues to credit the source, while the platform documentation of both vendors treats the values it observes as chosen by an untrusted party. What the credit reflects is the threat model of the Linux RNG, which does not assume a hostile host, rather than any assessment of the source under confidential computing conditions.

Disk I/O timing and the hardware RNG framework fall under the same finding. Section 5.2.2 states that I/O remains under hypervisor control in TDX 1.0 and that entropy reaching a guest through an emulated device path originates outside the trust boundary. For virtio-rng this is decisive, since the values it delivers are produced on the host and passed through a device the hypervisor emulates. The source was already assessed as fully host-dependent in Section 4.4.1, and confidential computing confirms that assessment without altering it. TEE-I/O, introduced with TDX 2.0, extends the isolation boundary to directly attached devices and could change this conclusion for some device classes, though it is not yet widely deployed and lies outside the scope of this work.

Bootloader entropy is affected by the same limitation, since the virtual firmware that supplies it is provided by the hypervisor. The recommended kernel command line for TDX guests reflects this directly by setting `random.trust_bootloader` to `n` [7, Kernel command line], so that firmware-supplied entropy is mixed into the pool without advancing the entropy estimate. The setting is recommended, not enforced, but it documents the platform vendor's own assessment of the source.

HID event timing remains absent, as confidential computing changes nothing about the hardware complement of a server guest. Device identifier injection remains available and remains uncredited, and the identifiers it mixes in continue to be assigned by the host.

VM fork detection occupies an unusual position. The failure mode it addresses arises when a guest is cloned or restored from a snapshot, and in current confidential computing

deployments those operations are largely unavailable. Guest memory cannot be streamed to another host without the hypervisor access that SEV-SNP and TDX exist to prevent, and the common virtualization stacks therefore document snapshots, save and restore, and live migration as unsupported for protected guests [53]. SEV-SNP defines a Migration Agent for this purpose [54], and at least one cloud provider has implemented live migration for confidential VMs [55], which indicates a restriction of implementation and not of architecture. Where migration is supported, the failure mode returns, and the mechanism that guards against it continues to depend on a VM Generation ID that the hypervisor maintains. The protection is therefore absent where it is not needed and hypervisor-dependent where it is.

The auxiliary sources require no separate treatment. Scheduler-based collection runs only while the pool remains unseeded and ceases once the CRNG reaches the fully seeded state, so that it bears on the boot interval alone, and for as long as it runs it observes events occurring within CPU time the hypervisor allocates and is mediated for that reason. Power management events are marginal on server workloads and advance no entropy estimate. The Latent Entropy GCC plugin contributes a compile-time seed and, through a second path invoked after each initcall, a further uncredited contribution at runtime. Both derive from the kernel image and the boot control flow, so an operator who supplies the image is in a position to reconstruct them.

RDRAND and RDSEED are the sources whose assessment confidential computing changes. Sections 5.2.1 and 5.3.1 described the mechanisms. Under TDX the TDX Module holds exclusive authority over a Trust Domain’s execution controls and state-save areas, and under SEV-SNP the VM Save Area

is encrypted and marked owner-exclusive in the Reverse Map Table. In both cases the hypervisor loses the two capabilities the interception attack required, namely the ability to configure exiting for these instructions and the ability to write into the guest’s saved register state. The instructions move from interceptable to protected, and Section 5.5.2 examines what the platform makes of that.

Two observations follow from the table. The first concerns the scope of the change. Of eleven sources, confidential computing alters the assessment of one. The count of eleven is in one respect generous, since four of those entries advance no entropy estimate at all, device identifier injection, VM fork detection, power management, and the Latent Entropy plugin being divergence and reseed mechanisms rather than entropy sources in the narrow sense. A fifth, HID event timing, is absent from a server guest altogether and so credits nothing in practice. Of the six entries that actually credit entropy in the environment under study, exactly one remains unknown to a malicious host. Every other source remains where Section 4.4.1 left it, because the guarantees of TDX and SEV-SNP are directed at memory and register state, not at the event stream a guest observes. Neither architecture falls short in this respect. A protected guest still requires scheduling, device emulation, and interrupt delivery, and the party performing those functions necessarily determines what the guest sees when it measures them.

The second observation concerns the relationship between crediting and trustworthiness, which the table records separately because the two diverge. The kernel credits several sources whose values a confidential guest has no reason to trust, most consequentially interrupt timing. This divergence follows from layering and does not indicate an oversight. The

TABLE III
REASSESSMENT OF THE SOURCES UNDER CONFIDENTIAL COMPUTING.

Source	Status in conventional VM	Change under CC	Credited (default)	Unknown to malicious host
add_interrupt_randomness	Mediated, reduced	Classified as host-controlled threat	Yes	No
add_disk_randomness	Mediated, strongly reduced	I/O remains hypervisor-controlled	Yes	No
add_input_randomness	Absent	none	n/a	n/a
add_device_randomness	Mediated, no credit	Identifiers assigned by host	No	No
add_hwgenerator_randomness	Mediated (host-supplied)	None in TDX 1.0, TEE-I/O out of scope	Yes (driver quality)	No
add_bootloader_randomness	Mediated (virtual firmware)	Not credited under recommended TDX config	Yes (Y by default)	No
add_vmfork_randomness	Depends on VM Generation ID	Failure mode largely absent, snapshots unsupported	No (triggers reseed)	No
Scheduler-based	Mediated	Scheduling remains with hypervisor	Yes	No
Power management	Marginal, mediated	none	No	No
Latent Entropy GCC Plugin	Compile-time seed plus uncredited runtime path	none	No	No (derivable from image)
RDRAND / RDSEED	Available, interceptable	Interception prevented by construction	Yes	Yes, within architectural scope (5.6)

Linux RNG operates under a threat model in which the host is not adversarial, and the entropy accounting inherited from that model does not distinguish between a source that is unpredictable and one that is merely unpredictable to the guest. Configuration can narrow the gap, as the recommended TDX command line does for bootloader entropy, but it cannot close it, since the sources in question cannot be excluded without removing the guest’s ability to seed at all under configurations where the CPU source is distrusted.

Read across the three assessments, the trajectory shows narrowing and no recovery. Section 3.2.3 enumerated eleven sources without qualification. Section 4.4.1 found one that virtualization does not mediate. Confidential computing protects that same one and leaves the remainder as they were. The entropy landscape of a confidential guest is therefore no broader than that of a conventional guest, only more sharply defined, and the following section turns to the source that survives the narrowing.

5.5.2 RDRAND and RDSEED as the Mandated Primary Source

Section 5.5.1 established that confidential computing alters the assessment of exactly one entropy source. That change concerns more than degree. In Chapter 4 the concentration on RDRAND emerged as a finding of the analysis, a conclusion drawn about an architecture that had not been designed with that concentration in mind. Under confidential computing the platform writes the same concentration into its design. This section examines the content of the mandate, the mechanism that implements it, and what the decision to impose it reveals about the trust model of confidential guests.

The requirement is not specific to Intel. It is implemented in the architecture-independent confidential computing layer of the kernel and evaluated for any platform the kernel identifies as a confidential guest, so it applies to SEV-SNP guests on the same terms as to Trust Domains. Section 5.2.2 discussed it in its TDX context, and the mechanism itself is architecture-neutral. Its author is Jason Donenfeld, the maintainer responsible for the redesign of the Linux RNG described in Section 3.2, and the reasoning given in the commit restates the argument of Section 4.4.3 in the compressed form appropriate to a commit message. Donenfeld’s point is that a malicious host can modify or observe nearly all inputs to a guest, which leaves RDRAND as the only remaining source of entropy for confidential guests [56]. A conventional system responds to a failing hardware RNG by continuing to gather entropy elsewhere. A confidential guest has nowhere else to gather from, which makes such a failure catastrophic at boot, where the pool is first seeded, even though its consequences later in the lifetime of the system are more theoretical.

One implementation detail qualifies what the mandate actually guarantees. The 256 bits obtained at boot are contributed through `add_device_randomness` [56], the same function Section 3.2.3 identified as a path into the pool that carries no entropy credit. The commit therefore guarantees that RDRAND material is present in the pool without causing that material to advance the entropy estimate. Crediting remains

governed by `random.trust_cpu` as described in Section 3.2.2. Donenfeld gives the reasoning explicitly. Any driver may contribute seed material of any quality through this path, since such a contribution can only improve the quality of the pool or leave it unchanged and can never degrade it. That property makes it the appropriate mechanism for a platform feature that should not be entangled with the accounting logic of the RNG proper. Mandate and crediting decision are thus separate questions, and a guest configured with `trust_cpu` set to N would still receive the mandatory contribution while declining to count it.

The mandate draws its force from the failure behavior. If the boot-time contribution cannot be obtained, whether because RDRAND fails or because the platform provides no such instruction, the kernel calls `panic` [56]. A confidential guest does not boot without it. This is a severe response by the standards of the RNG, which is otherwise designed to degrade gracefully when a source becomes unavailable, and the severity follows directly from the concentration Section 5.5.1 documented. Graceful degradation presupposes an alternative to degrade to, and a confidential guest has none. By aborting instead of continuing, the kernel acknowledges in the boot path itself that the redundancy on which the combiner construction depends is absent from a confidential guest.

The mandate also inverts a position the Linux community arrived at over a decade earlier. Section 3.3.3 described the response to the Snowden disclosures, in which the proposal to remove RDRAND from the kernel was settled on the terms that the instruction would be used, but never alone, and never as the sole basis for seeding. The reasoning was that a proprietary hardware source of uncertain provenance should not be trusted without corroboration from independent software sources. Confidential computing reverses the relation. The software sources are now the ones assumed to be under adversarial control, and the hardware source supplies the corroboration that remains. This follows from a changed threat model and does not retract the earlier concern, to which Section 5.6 returns. The adversary was once a distant manufacturer of uncertain intent, and it is now the operator of the machine on which the guest is running, so the nearer adversary takes priority over the more remote one.

The concentration is at its most complete during the first boot of a newly created guest. Section 4.4.2 examined that interval and found it exposed for two reasons that reinforce one another, and both are sharper in a confidential guest. No stored seed exists, since the guest has never shut down. Firmware entropy is present but uncredited under the recommended configuration. The software sources have accumulated nothing, and Section 5.5.1 places them outside the trust boundary in any case. `virtio-rng`, where available at all, originates on the host. Every path to the pool except the CPU instruction is either empty or untrusted, and this is the same interval in which SSH host keys and comparable long-lived material are generated. The narrowing traced across three chapters therefore reaches its endpoint in the interval whose consequences last longest.

A closer reading of the implementation qualifies the mandate further. The Intel specification uses RDRAND as a shorthand for the CPU hardware entropy instructions collectively, and the ordinary seeding path described in Section 3.2.4 reflects that generality, since it attempts RDSEED first and falls back to RDRAND only where the former is unavailable. The confidential computing mandate does not follow that pattern. Its loop calls the RDRAND primitive alone and never attempts RDSEED, in the same kernel that prefers RDSEED everywhere else it seeds the pool. The mandate is literally what its name says.

This matters for what the two instructions guarantee. Section 3.3.2 established that RDSEED provides a full-entropy guarantee across successive outputs while RDRAND does not, its output being an expansion of a shared DRBG state. Where several sources contribute, an absent independence guarantee in one of them is of limited consequence, since the combiner draws on whichever contribution is strongest. Once a single source carries the entire guarantee, the property that successive outputs are independently seeded from fresh physical entropy stops being a refinement and enters into what is being relied upon. The instruction the platform mandates for that role is the one that does not provide it.

One consequence of the mandate deserves to be stated alongside it, since it hands the host a lever it would otherwise not have. The kernel takes the instruction to be available on the evidence of CPUID, and under SEV-SNP those values reach the guest through the hypervisor. A host that withdraws the instruction on that path does not corrupt the entropy a guest receives, it removes the only source the guest is permitted to rely on, and the mandate's own failure behavior then decides what happens next. Section 5.6 collects this path together with the other influence that survives the transition.

Section 5.5.3 states formally what the narrowing has done to the construction described in Section 2.2.3.

5.5.3 The Combiner Argument under Confidential Computing

Section 4.4.3 examined what becomes of the combiner guarantee when a hypervisor sits between the guest and the hardware, and found that virtualization violates the premise on which the guarantee rests. Most sources feeding a guest's entropy pool depend on the same actor for the events they observe, so a single adversarial position compromises many of them at once, and a construction whose security argument requires independent failure modes offers no defense against that. Chapter 5 invites the reading that confidential computing repairs this, since it places one source beyond the hypervisor's reach. That reading holds as far as it goes. It omits, however, that restoring the premise of the lemma and restoring the property for which one combines are two different things.

The formal position is straightforward. The lemma requires at least one contribution carrying sufficient entropy that the adversary cannot predict, and Section 5.5.1 established that RDRAND and RDSEED satisfy this condition inside a Trust Domain or a SEV-SNP guest. The combined output of the

input pool is therefore secure, provided the CPU instructions deliver what they are specified to deliver. A confidential guest is not in the position Section 4.4.3 described, where no contribution remained outside adversarial control.

The difficulty lies in what the construction was built for. Combining several sources serves principally to arrange that no single failure is fatal, so that a source which degrades, malfunctions, or falls under adversarial control can be absorbed by the others. Turning unreliable inputs into a reliable output is the lesser part of what it does. The primary property requires a plurality of trustworthy contributions, and a confidential guest has one. With a single reliable source the lemma is satisfied in the same sense that a sum of one term is a sum, and the guarantee it provides is correspondingly narrow.

Put precisely, the lemma states that the output is secure as long as one contribution is good, and it says nothing about what happens when that contribution is not. With several trustworthy sources contributing, the failure of any one leaves the guarantee intact and the only open question is which of the remainder now carries it. Where a single trustworthy source contributes, its failure removes the guarantee entirely, and no property of the combiner compensates. Nothing has changed in the construction, and the change is in the material it operates on.

This inference does not rest on theoretical argument alone, since the kernel makes the same assessment in its boot path. Section 5.5.2 described the panic that follows when the mandatory RDRAND contribution cannot be obtained. A random number generator designed to degrade gracefully, and which does degrade gracefully on every other platform, refuses to continue on a confidential guest. The reason is the absence of anything to degrade to. An implementation that still possessed usable redundancy would have no occasion to abort.

None of this makes the combiner superfluous under confidential computing, and three of its properties survive intact. Under a hypervisor that is not in fact adversarial, the mediated sources contribute genuine entropy and the construction absorbs their contributions in the ordinary way, even though the threat model treats them as compromised, treating a source as untrusted being a different matter from knowing it to be manipulated. The guarantee that a compromised contribution cannot degrade the output also holds unchanged, which matters precisely because a confidential guest continues to absorb inputs it has no reason to trust. The construction likewise remains the correct one should the composition of sources change, as it would under the TEE-I/O extension noted in Section 5.5.1. The mechanism therefore remains, and the plurality it was designed to exploit does not.

The security of the entropy pool in a confidential guest therefore rests on a single proposition, that RDRAND and RDSEED deliver what their specification claims. Confidential computing has narrowed the question raised in Chapter 4 more than it has answered it. A conventional guest left it unclear whether any contribution could be trusted, while a confidential guest makes clear which one must be. That shifts the burden

from the architecture, which Sections 5.2 and 5.3 showed to be sound in this respect, onto the source itself and onto the assurances available for it. Section 5.6 examines what those assurances cover and where they end.

5.6 Remaining Trust Gaps under Confidential Computing

Section 5.5.3 concluded that the security of the entropy pool in a confidential guest rests on a single proposition, and that the burden has moved from the architecture onto the source. This section examines what remains once TDX and SEV-SNP have delivered what they promise. The gaps that remain are of different kinds, and distinguishing between them matters more than enumerating them. Some are defects in implementations that can be corrected. Some follow from where the architectures draw their boundaries. One is untouched by confidential computing altogether, and it is the one on which the preceding analysis has come to rest.

The first kind is exemplified by the vulnerabilities Sections 5.2.2 and 5.3.2 documented. RMPocalypse exploited an incomplete fencing of the Reverse Map Table during platform initialization, and the security review of TDX conducted by Google identified ten confirmed vulnerabilities across eighty-one evaluated attack scenarios. In each case the architectural design was sound while its realization fell short. Defects of this kind are correctable, and both vendors have corrected the ones so far identified. They are also, unusually among security problems, detectable by the relying party, since the TCB version recorded in an attestation report identifies the firmware and microcode present on the platform, and a verifier can decline to provision secrets to a platform running a version known to be affected. Their discovery does not count against confidential computing. It indicates a technology receiving the scrutiny that a security architecture of this consequence requires, and the attestation infrastructure described in Section 5.4 provides the means to act on what the scrutiny turns up.

The second kind follows from the boundaries the architectures draw: availability, I/O, and interrupt delivery all remain hypervisor functions in both TDX and SEV-SNP, as the table in Section 5.5.1 sets out source by source. None of this constitutes an oversight. A virtual machine requires a manager, and the functions the manager performs determine what the guest observes when it measures its environment. Gaps of this kind cannot be patched, though extending the architecture could in principle narrow them, a possibility returned to below.

That this second kind of gap is more than a theoretical concern has been demonstrated directly. Radev and Morbitzer showed that a malicious hypervisor can reduce the entropy of the probabilistic kernel defences of an SEV or SEV-ES guest by intercepting CPUID and RDTSC, which are the instructions from which the early boot code draws its randomness before any entropy pool exists, and can thereby recover the KASLR offsets and comparable secrets [46]. The result is instructive here for what it leaves untouched, since the attack neither intercepts RDRAND nor defeats memory encryption, operating instead on a different input that the guest depends on at

the same moment and that the hypervisor still controlled in those generations. SEV-SNP has since narrowed both paths by validating CPUID responses against a page measured by the Platform Security Processor at launch, so that a manipulated value changes the attestation measurement, and by providing Secure TSC, under which the timestamp counter parameters are negotiated with the Secure Processor rather than set by the hypervisor [57]. The narrowing is nevertheless uneven, because CPUID enforcement belongs to SEV-SNP itself while Secure TSC is an optional feature whose kernel support arrived only recently, so that a guest running without it retains the exposure on the timestamp path.

One path of that kind remains open on current platforms and bears directly on the entropy question. The CPUID values a SEV-SNP guest observes are supplied by the hypervisor and filtered through the firmware policy before the guest sees them [52], [38, Sec. 15.36.23], so the host retains a say in whether the guest believes the instruction to be available at all. A guest that is told RDRAND or RDSEED does not exist will not call it, and no intercept is needed to bring that about. The same lever appears on the defensive side, where `clearcpuid=rdseed` is the mitigation recommended against CVE-2025-62626 on affected Zen 5 parts, which shows that withdrawing an instruction through the CPUID path is an established operation and not a hypothetical one. How far that path reaches in practice was not measured here. The guest measurements taken for this work cover the output of the instructions and not the CPUID surface through which their availability is announced, and the platform was released before that gap could be closed, so the path is established from the specification and the firmware interface alone.

The general lesson survives whichever generation a given deployment runs, and it qualifies the finding of Section 5.5.2 in a way that matters for this paper. The concentration documented there is a concentration of the sources the kernel accounts for, not of everything the guest consumes while it boots. Where the hypervisor retains authority over any input feeding that interval, the boot-time entropy of a confidential guest can be attacked without the protected source itself being touched, which is the same distinction between path and origin that the third kind of gap turns on.

The third kind differs in nature from the other two, and it is the one that matters for this work. Confidential computing establishes that the value a guest receives from RDRAND is the value the CPU produced. It establishes nothing about whether that value carries the entropy the specification claims. The guarantee covers the path and says nothing about the origin. Every concern raised in Section 3.3.3 therefore survives the transition to a confidential guest unchanged. The physical noise source remains inaccessible to external measurement, the ESV certification rests on statistical testing of data the manufacturer supplied and on design documentation the manufacturer wrote, and no physical protection perimeter exists around the entropy source that would permit the assurances a FIPS 140-3 Level 3 module provides.

CVE-2024-21977 illustrates that this gap is not hypothetical [58]. Incomplete cleanup following a CPU microcode update permitted a privileged attacker to degrade the entropy delivered by RDRAND, with a consequent loss of integrity for SEV-SNP guests. The case is instructive precisely because the isolation guarantee was not defeated. A guest in that situation receives from the CPU exactly what the CPU produced, the hypervisor intercepts nothing, and the attestation report is valid. The guest nonetheless receives degraded material, because the attack operated on the source itself and not on the path between the source and the guest. An architecture that secures delivery offers no defense against a source that delivers the wrong thing.

The trust reduction that motivates confidential computing deserves examination on this point, since Section 5.1 introduced it as a reduction from two parties to one. The observation is correct, and its implications warrant careful statement. A conventional deployment requires trusting both the CPU vendor and the operator of the hypervisor, while a confidential deployment requires trusting only the vendor, since the operator has been removed from any position of influence. The vendor, however, now occupies both sides of the arrangement. Intel guarantees through TDX that RDRAND cannot be intercepted, and Intel manufactures RDRAND. AMD guarantees through SEV-SNP that the register state cannot be modified, and AMD manufactures the entropy source whose output occupies those registers.

This is not circular in the logical sense, because the guarantee and the thing guaranteed address different adversaries. TDX protects against the hypervisor and not against Intel, and it makes no claim to the contrary. The reduction it achieves is real. It does not, however, strengthen the assumption that remains, and a reader who takes the reduction from two parties to one as an improvement in the quality of the residual trust would be mistaken. The remaining assumption is the same assumption that was already present, now carrying the full weight. The comparison with a hardware security module drawn in Section 2.3.1 provides the relevant contrast. An HSM introduces a second vendor from a distinct trust domain, so that a defect or a deliberate weakness on one side does not compromise the other. Confidential computing offers no equivalent. The comparison is nonetheless instructive without amounting to a recommendation, since the HSM arrangement carries its own practical objection in a cloud setting, access to the device being itself mediated by the party one is trying to exclude.

A further limitation lies in the accounting and not in the architecture. The kernel permits a guest to decide, source by source, which contributions advance the entropy estimate, and Section 5.5.1 noted the recommended TDX configuration that credits the CPU source while declining to credit firmware entropy. These decisions are made independently, and nothing in the mechanism relates them to one another. An operator who distrusts the CPU source and sets `random.trust_cpu` to N, while leaving `random.trust_bootloader` at its default, may credit firmware-supplied bytes at full value that the firmware obtained by executing RDRAND [59]. The kernel

cannot detect this, since it receives a buffer of random data with no indication of provenance, and no such indication exists to be provided. The entropy accounting treats sources as independent because it has no means of establishing otherwise. The concern of Section 4.4.3 thus reappears at a different layer. There the failure of independence was a property of the threat model, and here it is a property of the bookkeeping.

Whether the second class of gaps is as fixed as it appears remains an open question, and current work suggests it may not be. The Secure VM Service Module introduced by AMD runs at the highest privilege level within a confidential guest, isolated from the host and from the guest operating system alike, and is measured into the attestation report [60]. Its purpose is to provide services that a guest would otherwise obtain from the hypervisor, relocating them inside the trust boundary instead of trusting the party outside it. Section 5.3.1 described the related Paravisor construction that Azure employs for the same end. In its deployed form the module offers little beyond an ephemeral virtual TPM, and it currently runs only on SEV-SNP platforms. Its development plan is nonetheless explicit about a wider ambition. It defines a paravisor mode in which the module handles the exceptions a confidential guest would otherwise process itself and emulates security-sensitive devices on behalf of the host hypervisor, and it lists support for Intel TDX and for delivering interrupts to lower privilege levels among its planned work items [61].

That last point bears directly on this chapter. Interrupt timing is the most productive software entropy source available to a guest, and Section 5.5.1 excluded it on the ground that its delivery is a hypervisor function. Should interrupt handling move inside the trust boundary, the exclusion would require revisiting, and with it the finding that confidential computing alters the assessment of exactly one source. The concentration this chapter has traced would then belong to a particular stage of the technology and not to confidential computing as such. That stage is the present one, and the analysis above describes it accurately, but the boundary has proven movable before and may move again. Chapter 9 returns to this as a direction for further work.

Of the three kinds of gap, two admit of remedy. Implementation defects are corrected as they are found, and their presence or absence is visible to a relying party through the attestation report. Architectural boundaries are fixed for a given generation of the technology and are already being contested by the service module work described above. The third admits of no remedy from within confidential computing, since it concerns what the protected source produces and not what happens to its output afterwards. It is also the gap on which everything else now depends, because Section 5.5.3 established that a confidential guest has no second source to fall back on. Chapter 6 accordingly turns to the only form of examination available to a party outside the manufacturer, namely measurement of the output the instructions deliver, and to what such measurement can and cannot establish.

6 ENTROPY ASSESSMENT METHODOLOGY

6.1 Entropy Assessment Frameworks

Estimating the entropy of a hardware random number generator whose internals cannot be observed requires a framework built for exactly that purpose. NIST SP 800-90B is that framework, and two requirements make it the only viable choice for this paper.

The first requirement is methodological appropriateness. SP 800-90B was designed specifically for the assessment of physical entropy sources [11], which are sources that are not guaranteed to be independent and identically distributed and whose internal state cannot be directly observed. Both RDRAND and RDSEED fall squarely into this category. Their outputs pass through conditioning and, in the case of RDRAND, a full DRBG layer before reaching the caller, the underlying physical noise source is inaccessible to any external measurement, and the hardware is subject to thermal memory effects and CPU-load-induced correlations that make the IID assumption unjustifiable. Alternative test suites such as NIST SP 800-22 [62] or Dieharder [63] are statistical randomness tests that detect gross failures in output distribution but do not estimate min-entropy and were not designed for physical noise source evaluation. Applying them here would answer a different question than the one this paper asks.

The second requirement concerns the standing of the method rather than the comparability of its outputs. SP 800-90B is the procedure that manufacturers apply to their own sources and that the CMVP accepts when it issues an entropy certificate [12], [64], so assessing RDRAND and RDSEED under it means assessing them by the standard the industry has agreed on for exactly this purpose. What the certificates report is nevertheless a different object from what this paper measures. They characterise the raw output of a noise source before conditioning, whereas an instruction returns output that has passed through conditioning and, in the case of RDRAND, a DRBG. The certified figures are therefore context for the assessment and not a baseline it is measured against, and the question of what an assessment of conditioned output can establish at all is itself one of the questions this paper puts, as the measurement family described in Section 6.3.2 makes explicit.

The assessment suite is implemented as an open-source C++ package maintained by NIST under the identifier `usnistgov/SP800-90B_EntropyAssessment` [65], in the build Section 6.2.6 identifies. It requires a GCC-compatible C++11 compiler with OpenMP support and depends on the `bzlib`, `divsufsort`, `jsoncpp`, `GMP`, and `GNU MPFR` libraries. The suite compiles into four separate binaries covering IID testing, non-IID entropy estimation, restart testing, and conditioning analysis, of which the non-IID binary and the restart binary are the operative tools for this paper.

The non-IID track is used for all sequential entropy estimates in this paper. SP 800-90B requires that any source whose samples cannot be demonstrated to be IID must be assessed on the non-IID track, and the choice made here is to assume

nothing in that direction. The IID property would have to be demonstrated before the simpler single-estimator analysis could be applied, and demonstrating it for a source whose internals cannot be observed is not something an output-level test can accomplish. No IID assessment was therefore run, and none is required for the track this paper uses. It should be noted that the expectation is not that these sources would fail such a test. Conditioned output of the kind RDRAND returns is likely to pass it, which is precisely why passing would establish little, since the property being tested is one the conditioning stage produces rather than one the noise source possesses. Treating the source as IID on that evidence and applying only the simpler analysis would systematically overestimate entropy and produce unsound security claims.

The non-IID track applies ten distinct estimators simultaneously to the same dataset, each modeling the output source under a different assumption about how an adversary might predict the next symbol. The Most Common Value and Collision estimators assess bias and symbol-level predictability directly from output frequencies. The Markov and t-Tuple estimators model sequential dependencies by examining transition probabilities between symbols and the frequency of short symbol sequences respectively. The Longest Repeated Substring estimator searches for the longest sequence that appears more than once in the dataset, which is a direct measure of exploitable repetitive structure. The four prediction-based estimators simulate adversaries using adaptive strategies:

- the MultiMCW estimator tracks the most common symbol within sliding windows of multiple sizes,
- the Lag estimator tests whether past outputs have periodic predictive power over future ones,
- the MultiMMC estimator maintains multiple Markov models simultaneously and selects the best-performing one at each step, and
- the LZ78Y estimator applies an adaptive dictionary compression strategy to predict unseen sequences.

The Compression estimator estimates entropy through the compressibility of the bitstring, treating high compressibility as evidence of low entropy. The result of the non-IID assessment is the minimum across all ten estimator outputs, reported as a lower bound rather than a point estimate and reflecting the conservative design philosophy of the standard. In a security context it is preferable to underestimate available entropy than to overestimate it. For the eight-bit symbols used here the suite runs the estimators twice, once over the symbols and once over the bitstring derived from them, and the value that is reported is the smaller of the symbol-level result and eight times the bit-level result. Which of the two branches binds is a property of the data rather than of the method, and it varies across the runs collected for this work, so the branch is recorded for every run and reported with the results.

This conservative design has a well-documented cost. Academic analysis has shown that two of the ten estimators, the Collision Estimate and the Compression Estimate, can produce significant underestimates for sources whose output is nearly uniform or whose distribution departs from the

IID assumption in specific ways. Zhu et al. demonstrated in 2017 that both estimators underestimate entropy for perfectly random IID sources [66], with the Compression Estimate dominating the minimum result approximately 73 percent of the time on such data and consistently producing values well below the true min-entropy. The underlying reason is that compression algorithms capture local sequential patterns rather than the global distributional property that entropy actually measures, which means the Compression Estimate can penalize a genuinely high-entropy source that exhibits structure at the scale the algorithm targets. NIST itself acknowledged during the public comment period [67] for SP 800-90B that the Collision and Compression estimators tend to produce the lowest values among the ten and that these values may not accurately reflect the true entropy of a well-behaved near-IID source. The effect is visible in the measurements taken here, although not in the shape the literature would suggest. A source of known full entropy does not assess at eight bits per symbol but at a floor in the region of 7.47, so the underestimation is real and reproducible. Which estimator produces that floor varies from run to run, and the Compression Estimate is one among several rather than the dominant one, so the specific attribution in Zhu et al. does not carry over to this data unchanged. The direction of the error also fails to be uniform. Against a source that was constructed to lack entropy, the assessment does not underestimate but overestimates, in every run of the ladder in Section 6.3.2 and by as much as a fifth of the true value, which is the reason the results reported here cannot be described as conservative in the correct direction without qualification.

The restart test, implemented in the `ea_restart` binary, requires a dataset in a distinct format consisting of a matrix of 1000 rows by 1000 columns [11, Sec. 3.1.4], where each row represents one restart of the entropy source and the 1000 columns represent the first 1000 output samples collected immediately after that restart. The measurement reported here uses 535 rows rather than 1000, for reasons of platform access that Section 6.3.6 sets out, and is therefore an analogous test over a reduced matrix rather than the standard one. The reduction rules out the reference implementation, which accepts a dataset of exactly one million samples and holds both dimensions fixed in its source, and whose sanity check draws its bound for a thousand samples per row while treating rows and columns alike. A matrix of 535 by 1000 is not square, so the two dimensions need separate bounds. The analysis is therefore carried out directly on the matrix, as Section 6.3.6 describes, using the procedure the standard defines rather than the binary that implements it for the standard shape.

A second qualification applies to what a restart dataset can capture at all, and it is a property of the platform rather than of the collection. The test is aimed at the first outputs a source delivers after a reset, but firmware and kernel initialisation draw on the instruction long before any collector can run. What is recorded is therefore output number n after the reset, for an n that is unknown and varies between boots, rather than output number one. The question the test asks, whether the source returns to a predictable state after a reset, is answered

only to the extent that such a state would persist that far into the output stream. The row datasets test whether the source delivers consistent entropy within a single restart sequence, while the column datasets test whether the first sample after a restart, the second sample after a restart, and so on across all 1000 restarts show predictability correlated with the restart event itself, which would indicate that the initial state of the source is partially determined by factors that repeat across restarts. This is the test that examines whether the hardware source returns to a predictable state after a reset, which is a different question from the boot-phase weakness of the Linux pool that the BSI 2016 study documented and that the measurement family in Section 6.3.5 addresses. The methodology for collecting the restart dataset on the bare-metal platform used here is described in Section 6.2.

One practical constraint of the suite is relevant to the design of the data collection in Section 6.2. The suite loads the full dataset into memory before analysis begins and is not designed for streaming operation, with the LRS estimator in particular building a suffix array over the entire input that requires working memory substantially larger than the dataset itself. SP 800-90B requires a sequential dataset of at least one million samples for entropy estimation, and provides that for non-binary data the additional bitstring-level assessment may ignore bits after the first one million [11, Sec. 3.1.3]. This paper uses 10^7 eight-bit samples per measurement run, which exceeds that minimum by an order of magnitude, occupies ten megabytes on disk and about one gigabyte of working memory during the assessment, measured as a peak resident set of 1,042,008 kilobytes for an ordinary stream, and provides sufficient statistical power for all ten estimators to produce stable estimates. A stream that repeats at a long period costs more than twice that, since the tuple estimates grow with the repetition, and the datasets of Section 6.3.2 reach about 2.2 gigabytes.

The choice of that size rather than a larger one follows from what a floor produced by the estimators actually depends on. A calibration run over data of known full entropy does not return eight bits per symbol, and it does not return one value either. Which of the two branches binds decides where it lands, so that runs over data of identical construction spread between about 7.34 and 7.55 bits per byte at this length. The floor is therefore a description of the procedure rather than a threshold a dataset can be held to, and Section 7.2 shows what happens to anyone who treats it as one. A run of 10^8 samples returns a value that differs from these by about 0.040, and that difference is systematic rather than random, because at the larger size a different estimator on a different branch produces the minimum. A floor obtained at one sample size is therefore not a statement about the source that can be carried over to another sample size. Calibration and measurement are consequently run at the same size throughout, which is what makes the calibration usable as a basis for interpreting the measurements at all. Repeated runs use independently collected data rather than different portions of one collection, since `ea_non_iid` is deterministic and would otherwise reproduce its own result.

SP 800-90B is accompanied here by two further procedures, applied not as independent confirmations of the same quantity but as additional detectors placed over the ladder of sources described in Section 6.3.2, where what the source contains is known in advance and the question is what each procedure notices. The second of them is BSI AIS 20/31, the assessment methodology developed by the German Federal Office for Information Security and used within the Common Criteria evaluation scheme. Version 3.0, published in September 2024, replaced the two-procedure structure of the 2011 edition with a single black-box test suite, designated T_{irn}, which a physical generator must pass. The suite consists of four tests, a monobit test and a poker test over 20,000 bits each and the MultiMMC and LZ78Y prediction estimates over one million bits each, and it consumes 2,040,000 consecutive bits in total [22, Sec. 4.6.4]. The two prediction tests also yield a min-entropy estimate per bit, against which the standard sets a threshold of 0.95, while the remaining tests return a verdict alone. The original AIS 31 suite was released by the BSI in 2004 [68] as a Java implementation and is publicly archived, and the open-source implementation following version 3.0 [69] exists as a Windows build. Section 6.2.6 sets out how the suite was applied here and what was validated against that build. Because this paper targets BSI submission, its results are a practical requirement that reviewers will expect to see alongside the NIST assessment. What they add to it is less than the pairing suggests, since version 3.0 adopts two of the NIST estimators for its own entropy statement, so the second framework inherits the blind spot of the first rather than covering it. Both suites are applied to the same collected bitstreams without requiring separate data collection.

The third element is a compression check applied to each collected bitstream, using xz in its strongest preset with a single thread, invoked as `xz -9e -T1`, and recording the ratio of compressed to original size alongside the formal results. That preset works with a dictionary of 64 MiB, and the dictionary size is stated here because it decides which period lengths remain visible, since structure that repeats at an interval longer than the dictionary is no longer available to the compressor. The periodic source on the ladder in Section 6.3.2 repeats every 625,000 bytes and therefore falls well inside that window, which is the reason the compression check sees a dataset that both of the other procedures miss.

The division of labour between the two is sharper than a cross-check would suggest, and it can be stated with a bound. On a periodic stream the estimators return the correct answer, close to zero, wherever they terminate, and whether they terminate depends on the period rather than on the size of the dataset. Over a series of such streams, periods up to 16,384 bytes completed within nine minutes at both one million and ten million symbols, and each returned an estimate of zero, while a period of 32,768 bytes and above did not complete within the thirty minutes allowed, at either size. The boundary between the two therefore lies between 16 and 32 kilobytes of repetition length on the platform used here. The compressor is unaffected by this and reduces each of these streams to slightly more than one period. Where the formal assessment stops

being usable, therefore, the informal check becomes trivial, which is the reason both are carried and the reason neither is treated as a confirmation of the other. On a calibration run of ten million bytes the result is 10,000,560 bytes, a ratio of 1.000056, which is what incompressible data looks like under this measure. Incompressibility is a necessary condition for high entropy because a source whose output is significantly compressible exhibits exploitable structure by definition, and a compression ratio approaching one confirms the absence of obvious low-level structure even when the formal estimators already report near-maximum values. The compression check does not produce a min-entropy estimate and is not a substitute for either formal assessment, but it provides an immediately interpretable first-order indicator that is free of the distributional assumptions embedded in both the NIST and AIS 31 estimators and is the same informal cross-check that the BSI 2016 study applied to its own bitstreams [3, Sec. 5].

One boundary of the assessment sits before the estimators rather than among them. A stream consisting of a single repeated symbol is rejected by the tool on reading it, with the note that the alphabet holds one symbol and no entropy is awarded, so the positive control of the ladder is caught by an input check and never reaches an estimator at all. That is the correct outcome, and it also marks how little of the detection the ten estimators are responsible for at the extreme end.

Agreement between the three carries weight only where the truth is already known, which is the situation the ladder in Section 6.3.2 constructs deliberately. There, agreement shows which defects the procedures catch and at what point they stop catching anything. Against an unknown source the same agreement establishes far less than it appears to, because all three operate on output alone, and a DRBG seeded with too little entropy produces output that none of them can distinguish from a well-seeded one. Where the three diverge, the divergence is reported and interpreted in the context of the known limitations of each method, and where they converge on a source whose entropy is not independently established, the convergence is reported as what it is, which is a statement about the procedures rather than about the source.

6.2 Experimental Platform

6.2.1 Access Levels and What Each Permits

The question this study asks concerns a party who rents compute and wants to know whether the entropy available to it can be trusted. That party has a bounded view of the machine it runs on, so the bound is part of the question. The measurements in Section 6.3 are organized accordingly, by the access a party would need to perform them.

Three levels are distinguished, named for the role that holds them. The tenant runs inside a virtual machine and holds no privileges over the host. Available to it are the instructions themselves, CPUID, the kernel log and command line of its own guest, the seeding state exposed under `/proc/sys/kernel/random`, and, in a confidential guest, the attestation report, among others introduced in Section 6.3. The firmware

TABLE IV
ACCESS EACH MEASUREMENT FAMILY REQUIRES, AND THE LEVEL AT WHICH ITS QUESTION ARISES.

Family	Access required	Level of the question
M1	tenant	tenant
M2	none (synthetic data)	tenant
M3a, M3c	hypervisor operator	tenant
M3d	tenant, with a hypervisor-operator counterpart	tenant
M4	tenant, controlled only at platform owner	tenant
M5	tenant	tenant
M6	platform owner	platform owner

configuration of the host, the behavior of the hypervisor, and the physical state of the machine all lie outside its view.

The hypervisor operator runs on bare metal and controls the virtual machine control structures of its guests. It configures instruction intercepts, emulates instructions, and observes every guest transition with exact timing. Chapter 4 assumes an adversary at this level.

The platform owner additionally holds firmware access and physical control. It sets the configuration recorded in Section 6.2.4, performs real power cycles, and runs on hardware shared with no other party.

The table below records the level each measurement family requires and the level at which its question arises.

Two rows describe what was intended rather than what was carried out. M3c could not be performed because the Intel platform exposed no path to creating a Trust Domain, and M3d was not performed because the machine was released before the guest could be measured on the CPUID path. Both are recorded in Section 6.5. M3a was carried out twice, once on the rented Intel machine and once on a local client machine, and the second of these is what made the instruction-level work independent of the rental.

Three of the entries state a condition in place of a level and are resolved here. M2 requires no measurement hardware, since its datasets are constructed rather than collected. M3d runs inside a guest and observes what the guest is told about the instructions available to it, and establishing that condition requires a corresponding action on the host. M4 runs inside a guest, where demand from other tenants on the same die cannot be separated from the load under study, and the controlled measurement therefore requires exclusive hardware.

Three properties of this arrangement bear on the argument, and Section 6.5 returns to each once the results are available.

The tenant and the adversary occupy different levels. The party that consumes entropy sees less of the machine than the party that could interfere with it, and the gap between the two columns above follows from that asymmetry and is not a limitation of the experimental design.

Confidential computing leaves the boundary between the levels in place and adds a verification path at the tenant level. The attestation report grants the guest no new capability over the host. It allows the guest to check assertions about

its environment that would otherwise be accepted on trust. Whether a tenant can walk that path is decided by the image the provider supplies rather than by the tenant, as the guest measurements here found out, and Section 6.2.5 describes the same asymmetry from the side of the collected data.

The two rentals that supplied the platforms differ along exactly these levels, and the difference shaped what could be measured where. On the Intel machine the study held hypervisor operator access without firmware access, which permitted the interception work and ruled out both the firmware configuration and the power cycles. On the AMD machine it additionally held platform ownership, which is why the restart campaign and the firmware work in Section 6.2.4 exist on that side and have no counterpart on the other. The asymmetry in the results is therefore a property of what was rentable and not of what was interesting.

The level at which a measurement is possible and the level at which its question arises need not coincide. M6 requires power cycles and therefore platform ownership, and it answers a question a tenant is never in a position to ask. M3a asks whether a hypervisor can supply an entropy source with values of its choosing, which is a question about the tenant's position that only the hypervisor operator can settle. Most of the questions in the second column belong to the tenant, and few of them can be answered there.

6.2.2 Hardware Platforms

Two rented server machines carry the measurements, one from each vendor, and a local client machine carries the work that needed no server. All three are described here in the properties that bear on an entropy source, which are the processor generation, the number of cores competing for it, and the cache topology that determines how many of them share a path to it. The full records are in Appendix B.

The AMD platform is an EPYC 9354 of the Genoa generation, family 25, model 17, stepping 1, running microcode revision 0xa101158, with 32 cores and 64 threads on a single socket. Its 256 MiB of L3 cache is divided into eight instances, one per core complex die, which matters because a source placed behind the interconnect is shared by all of them while a source placed in each die would not be. The machine was rented from a provider that grants firmware access and a baseboard management controller, which is what made the firmware work and the restart campaign possible.

The Intel platform is a Xeon 6527P, family 6, model 173, stepping 1, running microcode revision 0x1000405, with 24 cores and 48 threads on a single socket and 144 MiB of L3 cache in a single instance. The machine supports Trust Domain Extensions and the host initialised them, which is the reason it was rented. It grants no firmware access, so its configuration was taken as delivered.

The client machine is a laptop with a Core i7-1255U, family 6, model 154, stepping 4, microcode revision 0x43b. It carries the interception work that does not depend on a server, the synthetic datasets of the detection ladder, and the measurement of what an instruction intercept costs in cycles. Using it for these is not a compromise, because none of them ask anything about server hardware. The interception question is about what a hypervisor can do to a guest, and the detection question is about what an assessment procedure notices, and both are answered wherever the software runs.

The asymmetry between the two servers runs through the whole of this chapter. The AMD machine was held for three days with full access, and the Intel machine for one day with tenant and hypervisor access. What follows from that is set out in Section 6.2.3, and what it costs the study is set out in Section 6.5.

6.2.3 Environments and the Asymmetry Between Them

An environment in this study is a machine together with the isolation boundary the measurement ran inside. Four of them appear, and the set is smaller than the original design intended.

Bare metal on both servers is the reference. Nothing mediates the instruction, the operating system is the only software between the measurement and the processor, and the figures taken here are what every other environment is read against.

A SEV-SNP guest on the AMD machine is the confidential environment. It ran with four virtual processors and eight gigabytes of memory under a firmware build for encrypted guests, and the guest kernel reported that memory encryption with SEV, SEV-ES and SEV-SNP was active. It supplied the baseline measurements of both instructions. What it did not supply is an attestation report, for the reason Section 6.2.5 gives, and what binds the datasets to it is weaker than the design provided for.

Conventional KVM guests carry the interception work. A minimal guest on the Intel machine received values of the operator's choosing in place of the instruction output, and a full distribution guest on the same machine showed what such a substitution does to a system that boots on it. Further guests on the client machine carry the same work independently of any rental.

No Trust Domain was measured. The Intel machine initialised the TDX host, the firmware reported the private key identifiers, and the kernel refused to create a guest of that type, since the hypervisor exposed no such virtual machine type and the attempt failed before any Trust Domain existed. Section 6.2.4 records the exact state. Renting a confidential guest from a cloud provider instead would have supplied a Trust Domain, and it was ruled out for reasons of time and because the

resulting environment would have been less inspectable rather than more, being one where neither the firmware nor the hypervisor is under the observer's control.

The consequence is an asymmetry that has to be stated rather than smoothed over. On the AMD side the study holds bare metal, a confidential guest and platform ownership. On the Intel side it holds bare metal and hypervisor access. Every claim about confidential computing on Intel in this work therefore rests on specifications and source code, and every claim resting on measurement inside a confidential guest is a claim about AMD.

6.2.4 Platform Configuration and Its Defaults

The AMD measurement platform did not arrive in a usable state. SEV-SNP required six firmware settings to be changed before the hypervisor reported the feature as available. This section records those settings, the dependencies between them, and the checks that confirm each one from the running system. The configuration is part of the measurement apparatus, so no dataset collected on this platform is interpretable without it.

SEV-SNP rests on five mechanisms that must all be active. They are configured independently, and a single inactive layer prevents the feature from initializing. Secure Memory Encryption Enable (SMEE) activates the AES engine in the memory controller. Every subsequent mechanism operates on ciphertext produced by this engine, and without it the hardware foundation stays inactive. SEV Control then permits the processor to assign per-guest encryption keys managed by the AMD Secure Processor. The SEV-ES ASID Space Limit partitions the 1006 available address space identifiers between plain SEV guests and guests using SEV-ES or SEV-SNP, and a limit of 1 leaves no identifiers for the latter, making both unusable regardless of other settings. SNP Memory Coverage allocates the Reverse Map Table, which records ownership and permitted guest physical address for every physical page. The table is allocated during POST and cannot be created afterward, so changes to this setting take effect only after a full power cycle.

The remaining two settings extend the integrity guarantee to device accesses. RMP enforcement occurs in the CPU path, so a device performing DMA would bypass it. The IOMMU must therefore be active in translating mode, and it must additionally advertise SNP support in its extended feature register. The Linux driver checks both conditions separately and refuses SNP initialization when either fails.

The following table records the settings as delivered and as configured for the measurements.

The settings were applied in two stages separated by a power cycle, which produced an intermediate state useful for verification. After the first four settings, the kernel reported SEV and SEV-ES as available and SNP as disabled, naming the IOMMU as the obstacle:

kvm_amd: SEV enabled (ASIDs 100 - 1006)

kvm_amd: SEV-ES enabled (ASIDs 1 - 99)

kvm_amd: SEV-SNP disabled (ASIDs 1 - 99)

TABLE V
FIRMWARE SETTINGS THAT HAD TO BE CHANGED BEFORE SEV-SNP WAS AVAILABLE.

Menu	Setting	As delivered	Configured
CPU Configuration	SEV Control	Enabled	unchanged
CPU Configuration	SEV-ES ASID Space Limit	1	100
CPU Configuration	SNP Memory (RMP Table) Coverage	Auto	Enabled
CPU Configuration	SMEE	Auto	Enabled
NB Configuration	IOMMU	Auto	Enabled
NB Configuration	SEV-SNP Support	Auto	Enabled

AMD-Vi: SNP: IOMMU SNP feature not enabled, SNP cannot be supported.

After the remaining two settings, the same checks reported the feature as active:

AMD-Vi: IOMMU SNP support enabled.

AMD-Vi: Extended features (0xa5bf7320a2294aee, 0x1d):
PPR X2APIC NX [7] IA GA PC GA_vAPIC SNP

ccp 0000:02:00:5: SEV-SNP API:1.55 build:49

kvm_amd: SEV-SNP enabled (ASIDs 1 - 99)

Each setting has an observable consequence that does not require access to the firmware setup. SMEE and SEV Control are confirmed by the Secure Processor reporting an SEV API version in place of the message that memory encryption is not enabled by the BIOS. The ASID Space Limit appears in CPUID leaf 0x8000001F as the minimum SEV guest ASID, which changed from 1 to 100, and in the partition reported by the hypervisor. RMP allocation is confirmed by the physical range printed at boot. IOMMU SNP support appears both as an explicit message and as the SNP entry in the extended feature list. The aggregate state is readable from the `kvm_amd` module parameters.

The availability of these checks has a consequence beyond convenience. A guest cannot inspect the firmware of the host it runs on, and none of the settings above are visible from inside a virtual machine. The verification chain remains available at an access level where the configuration itself is not, which places these checks among the observations a tenant can perform. Section 6.2.1 sets out the access levels this distinction refers to.

The configuration has two measurable consequences for the platform itself. The RMP occupies 768 MB, reducing memory available to the operating system from 184 GiB to 183 GiB. Enabling SNP also disabled the virtual APIC, reported by the IOMMU driver as a forced deactivation and by the hypervisor as a missing `HvInUseWrAllowed` capability on an SNP-enabled system. Interrupt handling on the host therefore differs between the measurements taken before and after configuration, and Section 6.5 records this as a limitation.

Instruction throughput was measured in both states as a control, and it was unchanged by the configuration. The absolute figures are not comparable across measurement campaigns, because the collection tool times the generating loop together with the hashing and writing of the collected data, and because a short run does not reach a steady rate. The same instruction

on this platform delivers 36.0 million values per second over a run of 34 ms, 37.5 million over 67 ms and 41.9 million over 596 ms. Rates are therefore reported here only alongside the length of the run that produced them. Active memory encryption and an allocated RMP impose no measurable cost on either instruction, which the analysis in Chapter 7 assumes and this measurement establishes.

Four of the six settings were delivered with the value Auto, and in all four cases the associated function remained inactive. SMEE is the clearest instance. SEV Control was already set to Enabled as delivered, yet the Secure Processor reported that memory encryption had not been enabled by the BIOS, and SEV became available only after SMEE was changed from Auto to Enabled.

The observation covers one system. Whether these values reflect the board vendor's factory defaults or a configuration applied by the hosting provider was not determined, and no claim is made about AMD platforms in general. Within that limit the finding is that confidential computing was not available on this machine as delivered, and that six settings across two menus were required to make it so. Chapter 8 takes up what follows from this for self-managed bare metal deployments.

The Intel machine has no counterpart to any of this, and the absence is itself the finding. No firmware access came with the rental, so the configuration could only be read as delivered and not set, and nothing about the state of the confidential computing features could be established below the level of what the operating system reports. What it reported was a host that had initialised Trust Domain Extensions, with the firmware announcing a range of private key identifiers and the module loading during boot. What it did not report was any way to use them. The hypervisor advertised the virtual machine types it supports without the bit for a Trust Domain among them, and an attempt to start one failed with the accelerator refusing the type, on the distribution kernel as well as on the unmodified vendor module. That is a platform boundary and not a result, since it says nothing about how a Trust Domain behaves and only that this rental could not produce one.

Read together, the two sides say something about what a tenant can verify. On the machine where firmware access came with the rental, six settings across two menus had to be changed before the confidential computing feature existed at all, and a guest on that machine could not have seen any of

them. On the machine where firmware access did not come with the rental, the same class of question could not be asked at all. A tenant is therefore in the position of the guest in both cases, and the attestation report is the only construct that speaks to it, which is what makes its absence from the guest measurements in Section 6.2.5 consequential.

6.2.5 Provenance Through Manifests and Attestation

A byte stream carries no record of the source that produced it. Two datasets collected under different conditions may be indistinguishable as data while supporting entirely different claims, and the analysis in Section 6.3.2 shows that statistical assessment does not recover the difference. Every dataset in this study is therefore accompanied by a manifest that binds it to the conditions under which it was collected.

The manifest records the identity of the physical machine. Board serial number and system UUID establish that the bare metal, KVM guest, and SEV-SNP guest measurements were taken on the same processor, which is the property the experimental design depends on and which no other field substitutes for. The same values allow a later rental to be checked against an earlier one. The records show this working in both directions, since the datasets of the AMD session all carry one board serial and those of the Intel session another, and an earlier feasibility test on a third machine is therefore not confusable with either.

Four version numbers are recorded, since each belongs to a layer that can be updated independently of the others. The kernel version identifies the subject of measurement. The BIOS version determines which processor capabilities are released to software. The microcode revision governs instruction behavior, including the behavior of RDRAND and RDSEED. The SEV firmware version identifies the code running on the AMD Secure Processor, which executes the SEV and SNP commands. A single version number for the system would leave three of these unrecorded.

Processor identity is recorded as family, model, stepping, and microcode revision, in place of the marketed product name. The mitigation for CVE-2025-62626 illustrates why. The kernel determines whether a processor is affected by matching family 0x1a against two model and stepping combinations and comparing the microcode revision against a minimum, and it disables RDSEED when the comparison fails. A claim about the entropy source that omits these four fields cannot be checked against such a rule.

The environment type, the full kernel command line, and the firmware configuration recorded in Section 6.2.4 complete the description of the measured system. The command line matters for M5, which varies `random.trust_cpu` and `random.trust_bootloader` across runs, and for any later attempt to reproduce a seeding time. Collection context adds a timestamp, the git commit of the collection tool, and a hash over the raw data. M4 additionally records thread count, thread placement, and the observed clock state, since the scaling result depends on all three.

For measurements taken inside confidential guests, the design provides for the attestation report to be stored alongside the manifest, since the report is signed by the platform firmware, binds the measurement to the environment that produced it, and carries the TCB version. Section 5.4 sets out the guarantees the report provides. No report accompanies the guest measurements reported here. The rented environment ran a distribution kernel without the guest attestation driver, so the device through which a report is requested did not exist, and the collection tool recorded the absence rather than substituting anything for it.

This has a further consequence for the guest datasets that has to be stated plainly, because the manifest does not carry it on its face. The environment field of a manifest is supplied by the operator rather than derived, and the manifests accompanying the guest datasets were written on the host, which is visible in them, since they carry the board identity of the physical machine, the host kernel command line, the full host topology and module parameters that exist only outside a guest. What binds those datasets to a confidential guest is therefore the report of the guest kernel that memory encryption with SEV, SEV-ES and SEV-SNP was active, together with the launch configuration as the operator recorded it after the fact. Neither is a signed statement, and the platform was released before the gap was noticed. Both are held with the datasets as a separate record, marked as reported after the fact rather than captured.

A separate block records what was loaded into the kernel, because a measurement taken under a modified hypervisor is a different measurement. For each virtualisation module it names the file, the package that owns it, the signature type and the version magic, and it records the kernel taint value alongside them, since a module built outside the distribution leaves a trace there that a file path alone would not show. Package ownership is resolved against the merged layout that current distributions use, where the module path reported by the kernel and the path recorded by the package manager differ. In every manifest collected with this tool the taint value is zero and every module is owned by a distribution package, so nothing outside the distribution was loaded while the measurements ran. The interception work is the deliberate exception, and it is not part of any measured stream.

The provenance of the collection tool itself is enforced rather than stated. The build records the commit it was built from and whether the working tree was clean, the recorded value is checked before a dataset is accepted, and a build from a modified tree marks every dataset it produces. This is what makes the gap on the Intel side visible in the first place, since those datasets predate the mechanism and say so.

The manifests are stored as one JSON document per dataset, held next to the raw data and under version control. The hash over the raw data is contained in the manifest, so a dataset and its description can be checked against each other after the fact.

What the manifest cannot do deserves as much attention as what it records. Almost every field is self-reported by the

system under measurement. Board and BIOS identifiers are read from SMBIOS tables supplied by the firmware, the microcode revision is read from the running processor, and the kernel command line is read from the kernel that was booted with it. An adversary who controls the environment also controls these values. For an honest collection the manifest is sufficient, whereas against the adversary assumed in Chapter 4 it is not.

The record is also not equally complete across the two platforms. The datasets from the Intel machine were collected before the collection tool carried its own provenance, so their records name no build of it, and the kernel they were taken under differs from the one the interception work on the same machine used. Nothing about them can be tied to a specific version of the software that produced them beyond what the manifest states, and the asymmetry this creates between the two platforms is part of what Section 6.5 sets out.

The attestation report is the one mechanism that would escape this, and it applies only to confidential guests. Its signature chains to a key the hypervisor does not hold, so an environment that carries one cannot assert a platform state it is not in. The architecture therefore offers an asymmetry in favour of the confidential guest, which is the one environment where provenance could be demonstrated rather than asserted. Realising it requires a guest image that can request a report, and in a rented environment that is a property of the image the provider supplies. The measurements reported here did not have it, so every environment in this study rests on self-reported provenance, and the environment that could have done better is the one that did not.

The collected datasets accompany this report, together with the collection tool and the analysis pipeline, in a repository held by the authors. The claim in Section 6.3.2 is a negative one, and a negative result invites the objection that the pipeline was misconfigured or that the modelled defect did not match the documented one. Handing over the data and the code allows that objection to be tested instead of argued. Datasets of RDRAND and RDSEED output with a recorded collection context are scarce in any case, and the fields above are precisely those a reader would need to relate such data to a specific processor erratum.

Section 6.3 sets out the fields each measurement family adds to the manifest, and Section 6.5 returns to the limits of self-reported provenance.

6.2.6 Analysis Pipeline

Every collected stream passes through the same three procedures, and the parameters of each are fixed here rather than chosen per dataset, since a parameter chosen after seeing the data is no longer a parameter but a result.

The entropy assessment uses the NIST reference implementation at commit 87c104d0 of 26 May 2026, which carries no release tag and is therefore identified by its hash. Streams are read as eight-bit symbols on the non-IID track, with the remaining options left at their defaults, so that each run reports the symbol-level estimate, the bit-level estimate and

the smaller of the symbol-level result and eight times the bit-level one. Runs are bounded at thirty minutes, and a run that reaches the bound is recorded as not terminating rather than as a low estimate, which is a distinction Section 6.1 shows to matter. One build produced every figure reported here, and it was checked against the earlier results of the collection platform before use, agreeing with them on every reported estimate and differing only in the far decimals of intermediate quantities, which is the expected consequence of a different floating point implementation.

Every measurement is accompanied by a calibration dataset of the same length, drawn from the operating system random number generator, and a result is read as a distance from what that calibration returns rather than from the theoretical maximum. Section 6.1 sets out why the two must share a length.

The compression check runs with the parameters Section 6.1 gives, at the strongest preset with a single thread and a dictionary of 64 MiB, and the recorded quantity is the ratio of compressed to original size.

The third procedure is the AIS 31 test suite `T_1rn` of version 3.0, applied to the same streams. The reference implementation of that suite exists only as a Windows build, and rather than port it, the suite was implemented directly against the constants of its published source at tag `v3_0_8`. Two of its four tests are counting tests over twenty thousand bits each and are computed here. The other two are the prediction estimates of SP 800-90B, sections 6.3.9 and 6.3.10, which version 3.0 adopts unchanged, and their values are taken from the same assessment run that produces the figures of the first procedure. The suite consumes 2,040,000 consecutive bits per attempt and allows a second attempt after a failure, and a test is passed at the thresholds the source fixes, a monobit count between 9654 and 10346, a poker statistic between 1.03 and 57.4, and a min-entropy of at least 0.95 bits per bit for each of the two prediction tests.

The implementation reproduces all four values of the reference report published with the tool, including the intermediate quantities of both prediction tests, against the sample input whose digest that report records.

Two properties of this arrangement bear on how the results of the suite should be read, and both are consequences of the standard rather than of the implementation. The first is that its entropy statement is not independent of the first procedure, since two of its four tests are the estimators of that procedure and therefore share whatever they fail to see. The second is that it examines 2,040,000 bits per attempt and at most twice that per dataset, which is about five percent of a stream of ten million symbols, so a defect whose period exceeds the examined window is invisible to it however well it is implemented. Section 7.2 reports a dataset on which exactly that happens.

The restart data take a different path, because their length and their shape both rule out the reference implementation for that test. Section 6.3.6 sets out the procedure that replaces it.

6.3 Measurement Families

6.3.1 M1 Baseline Min-Entropy Across Environments

M1 establishes what the two instructions deliver when nothing interferes with them, and it is the measurement every other family is read against. It runs on bare metal on both server platforms and inside the confidential guest on the AMD platform, covering RDRAND and RDSEED separately in each.

A run collects ten million eight-bit symbols in a single thread, written as the instruction delivers them. A failed call contributes nothing to the stream and is repeated, so the collected stream holds delivered values only and its length is fixed by construction. The rate of failed calls is recorded beside the stream rather than in it, since it describes the behaviour of the source and would otherwise be invisible in data that contains only successes. That separation matters on one of the platforms, where a majority of calls to one of the instructions fails even without competing load.

Three runs per instruction were collected in each environment that allowed it, each from an independent collection rather than from a division of one. The reason is that the assessment procedure is deterministic, so re-analysing parts of a single stream would reproduce its own result rather than test it. The Intel platform carries one run per instruction, because the rental ended before more could be taken.

The interpretation rests on the calibration described in Section 6.2.6, which fixes what the procedure returns for a source of known full entropy at the same length. What M1 can establish is therefore a distance from that floor and not an absolute quantity, and what it cannot establish is the subject of Section 6.3.2.

6.3.2 M2 Detectability of a Known-Defective Source

M2 turns the question around. Instead of asking what the assessment says about a source whose entropy is unknown, it asks what the assessment says about sources whose entropy is known in advance, and it does so on a ladder of constructed datasets that runs from a sound source to an obviously broken one. The datasets are synthetic by design, so no hardware is involved and the true min-entropy of each is available in closed form beside it.

The ladder has five rungs. The calibration rung is the output of a generator seeded from the operating system and is taken as carrying full entropy. The first defective rung reproduces the failure mode of CVE-2025-62626 by replacing a fraction of the thirty-two bit words with zero while signalling success, with the fraction varied over five values so that the point where the defect becomes visible can be located rather than assumed. The second defective rung is the one the argument turns on, a sound generator seeded with thirty-two bits of entropy, which produces output that is statistically indistinguishable from the first rung while carrying almost no entropy at all. The third repeats a recorded block at a fixed period. The fourth returns a constant value and serves as a positive control, since an assessment that does not flag it is misconfigured.

Every rung is put through all three procedures of Section 6.2.6, and the result is a detection matrix rather than a set of estimates. What the matrix is for is to establish where each procedure stops seeing a defect, and the three do not fail at the same place. The repetition rung has a further property that Section 6.1 describes, since the entropy assessment stops terminating above a repetition length while the compression check becomes trivial exactly there.

Each dataset records the seed it was generated from, so that the stream can be rebuilt bit for bit from the value stored beside it. That matters more here than elsewhere, because a negative result invites the objection that the construction was not what it was said to be, and a reader who suspects that can regenerate the dataset and check its digest.

The ladder is where the limits of the method are established, and the limits apply to every figure this work reports, including those of M1.

One property of the measurement platform belongs here rather than in the results. The ladder was collected on a Zen 4 processor, which is not the generation affected by the defect the first rung reproduces. The rung is a reconstruction of a documented failure mode and not an observation of it.

6.3.3 M3 Instruction Interception under Confidential Computing

M3 asks what a hypervisor can do to the instruction rather than what the instruction delivers, and it divides into four parts that are answered by different means.

The first reproduces the interception demonstrated in 2016. A modified hypervisor module enables the exit for the instruction and supplies a value of the operator's choosing in its place, and a guest on the same machine then reads that value with the carry flag indicating success. The reproduction was carried out on the Intel server and repeated on the client machine. Two different target registers were verified, which is the property that matters, since it shows the handler reading the destination from the instruction information field rather than assuming one. A second and independent trace of the same manipulation appears in the register dump the guest kernel prints when its own self-test fires, where the substituted constant is visible in both registers at the moment of detection.

The second part concerns AMD and is settled without an experiment. Neither the architecture manual nor the kernel source defines an intercept for either instruction, so there is no mechanism to enable and none to test. Section 4.3.3 sets this out, and the measurement design records it as answered by specification rather than left open.

The third part concerns a Trust Domain and could not be carried out, because the rented Intel platform exposed no way to create one. Section 6.2.4 records the state of that platform, and Section 6.5 counts the gap.

The fourth part concerns the path that survives interception, which is the announcement of the instruction to the guest through CPUID rather than the instruction itself. It was designed as a guest measurement with a corresponding action on

the host, and it was not carried out, because the platform was released before it could be. The path is therefore established from the firmware interface and the specification alone, as Section 5.6 sets out.

Two further measurements attach to this family. The first measures what an exit costs in cycles on a client processor, which replaces a figure this work otherwise takes from the literature. It uses CPUID as the trapping instruction, because CPUID exits unconditionally and therefore isolates the cost of the transition from any property of the instruction under study, and it subtracts the cost of the timing itself.

The second was to extend the interception question to RDSEED, whose exit is configured through a control bit independent of the one for RDRAND, and it was not carried out. The module that performs the interception was built against a kernel that no longer matches the running one, and loading a module whose version magic differs leaves the capability unset rather than producing an interception, which is the same failure this work met on the rented Intel platform. Whether RDSEED can be intercepted in the same way therefore remains open, and it cannot be inferred from the RDRAND result, since the two bits are configured separately and the specification treats them as independent.

6.3.4 M4 RDSEED under Contention

M4 measures what happens to the two instructions when more than one thread asks for them at once, which is the situation of a shared server and the condition under which the documented failure mode of RDSEED appears.

The family has four parts. A single-threaded baseline fixes the uncontended rate for both instructions. A scaling series repeats the measurement from one thread up to the full thread count of the machine, recording the delivered rate and the fraction of failed calls at each step. A placement series holds the thread count fixed and varies where the threads run, concentrating them on one core complex in one arm and distributing them across all eight in the other, which distinguishes a source placed behind the interconnect from one placed in each complex. That distinction can only be drawn on the AMD platform, since the Intel machine has a single cache instance and offers nothing to place threads differently across.

The fourth part reproduces the condition under which the defect in the Zen 5 generation was reported, which is sustained memory pressure, and records the number of draws made under that load alongside the result. The platform measured here is of the preceding generation, so the expected outcome is that nothing appears, and the value of the measurement is that the condition was applied rather than assumed to be absent.

The Intel datasets of this family were collected from the volatile system described in Appendix B, with an earlier build of the collection tool. Their rates are therefore not interchangeable with rates taken on the other platform, and the comparison that carries across is the shape of the curve rather than its height.

6.3.5 M5 Time to Fully Seeded

M5 measures the interval this paper began with, which is the time between the start of a system and the point at which its entropy pool is considered seeded, and it is the only family that measures the operating system rather than the instruction.

The first arm varies the two boot parameters that decide whether the kernel credits the CPU instruction and whether it credits the bootloader, and it repeats each of the four combinations over three real restarts. The measured quantity is the timestamp the kernel prints when the pool reaches the seeded state, read from the kernel log of each boot. Twelve boots is a small number, and it is what the interval allowed, but the quantity separates the configurations by two orders of magnitude, which is larger than the spread within any of them.

The second arm answers a different question, which is not when the pool is seeded but who draws from it before it is. A kernel built with the option that reports every access to the unseeded pool records each of them with its caller, and the resulting list is the consumer side of the interval that the first arm measures in seconds. Capturing the list requires an enlarged log buffer, which itself moves the seeding time, so the two arms are reported separately and the timing figures come from the first.

Three properties of the method bear on the result. The restarts must be real, since a restart that preserves memory carries a seed across and produces a seeded pool at a timestamp of zero, which is indistinguishable in the log from a pool seeded instantly. The filter used to find the relevant messages must not assume the state name of one kernel version, since that name changed. And the log buffer must be large enough to hold the reports, which on a boot that produces tens of thousands of them is far beyond the default.

The Intel platform carries a reduced version of this measurement, two boots from which only the seeding timestamp was preserved. One of them shows the pool seeded at a timestamp of zero, which is the signature the first property above describes, and the full log that would settle whether that boot preserved memory was not kept. The measurement is therefore reported as what it is, and Section 6.5 counts it among the limits.

One claim about the confidential case is worth separating from the measurement, because it is architectural and not observed here. The kernel aborts the boot of a confidential guest when the mandated contribution from the instruction cannot be obtained, as Section 5.5.2 describes. Setting the parameter that withholds credit does not trigger that abort, since the contribution is still mixed into the pool and only the accounting changes.

6.3.6 M6 Restart Tests

M6 is the restart test of the standard, applied to the hardware source rather than to the operating system. It asks whether the source returns to a state that repeats after the machine loses power, which is a question no tenant can ask, since it requires the power to be cycled.

The campaign ran on the AMD platform over three days under control of the management controller, which switched the machine off and on again through the power control interface rather than restarting it. Each cycle boots a system from an initial ramdisk, collects a row of one thousand values per instruction as early as the kernel permits, collects a bulk sample of about one million values behind it, records the boot identifier and the timing of the collection relative to the start of the kernel, and shuts down again. A cycle takes about three and a half minutes, most of which is firmware initialisation.

Five hundred and thirty five cycles were completed. The standard asks for one thousand, and the reason for stopping short is the access itself, since the management interface of the rented machine expired every two hours and had to be renewed for the campaign to continue at all. What is reported is therefore an analogous test over a reduced matrix, and the reduction is a property of the rental rather than of the method.

The reduction rules out the reference implementation, which accepts exactly one million samples, fixes both dimensions of the matrix in its source, and derives its sanity check bound once for a thousand values and applies it in both directions. A matrix of five hundred and thirty five rows by one thousand columns is not square, so each direction needs its own bound, and the procedure of the standard is therefore applied directly to the matrix. Two properties are examined. The first is whether any row repeats, which is what an intercepted or stuck source produces and what the pilot of this campaign shows in its extreme form, since an intercepted row holds two distinct values and repeats identically across boots. The second is the sanity check of the standard, with the bound simulated separately for rows and for columns.

Two properties of the setup bound what the family can establish. The timestamp counter of the processor restarts with the machine, so its value at the moment of collection distinguishes a cycle that removed power from one that did not, and the campaign reports that value for every cycle. A control arm that restarts without removing power was not run on this machine, so the distinction is argued from the counter rather than demonstrated against a warm restart. The second property is the one Section 6.1 describes as the n problem, which is that firmware and kernel draw on the instruction before any collector runs, so the collected row begins at an unknown point in the output stream rather than at its first value.

6.4 What the Results Are Compared Against

Three bodies of existing work stand behind the numbers this paper reports, and each of them is a different kind of comparison, which is why they are separated here rather than presented as one baseline.

The BSI 2016 study is the work this paper revisits, and the comparison with it is a comparison of statements rather than of figures. That study measured the entropy pool of an operating system and the behaviour of a virtual machine monitor, and it did not measure the output of the CPU instruction, whose entropy it took from design documentation. Its findings about the boot-time interval and about the feasibility of

interception are therefore the object of comparison, and its numbers are not.

One quantity does compare directly, and it is the interval measured by M5. The 2016 study reported the time its system needed to accumulate enough entropy after boot, and the same quantity is measured here on current hardware under a kernel whose entropy architecture has since been redesigned. The accounting behind the two numbers differs, as Section 4.4.2 sets out, so the comparison is between the length of an exposed interval and not between two values of the same variable.

The certificates issued through the validation programmes are the third body, and they are a contrast rather than a baseline. What they assess is the raw output of a noise source before conditioning, while what this paper measures is the conditioned output an instruction returns. The figures therefore describe different objects, and placing them side by side would suggest a comparison the data do not support. What the certificates do establish is what a vendor has had examined and what it has not, which is the question Chapter 5 treats as the residue of trust.

6.5 What This Methodology Cannot Establish

A study that reports what it measured owes an account of what it did not, and in this case the account is long enough to shape how the results should be read.

The two platforms are not equally covered. The AMD machine was held with firmware access and platform ownership for three days, and the Intel machine with tenant and hypervisor access for one. Every measurement that requires power cycles or firmware configuration therefore exists on one side only, and the Intel baseline rests on a single run per instruction rather than three. Where this chapter compares platforms, it compares what was measurable on each.

No confidential guest was measured on Intel. The rented machine initialised the host side of Trust Domain Extensions and exposed no way to create a Trust Domain, so every statement this paper makes about TDX rests on the specification and on published source code. The corresponding statements about SEV-SNP rest on measurement, and the two should not be read as carrying equal weight.

The confidential guest carries no attestation. The kernel of the rented environment had no driver through which a report could be requested, so the claim that those datasets come from a confidential guest rests on what the guest kernel reported about itself and on the launch configuration as the operator recorded it afterwards. The manifests stored with those datasets describe the host, because the tool that wrote them ran there.

Elapsed times are not comparable between environments. The collection tool measures the generating loop together with the hashing and writing of the data, and the storage path differs between a guest and bare metal. No throughput comparison between the two is reported, and the control measurement that

would have separated the effects was no longer possible once the platform was released.

The Intel datasets carry no tool provenance. They were collected before the collection tool recorded the commit it was built from, and under a different kernel than the interception work on the same machine.

The boot-time measurement on Intel is incomplete. Only the seeding timestamp of each boot was preserved, and one of the two shows the signature of a seed carried across a restart. Whether that boot removed power cannot be established from what remains.

The restart matrix is smaller than the standard requires. The campaign completed 535 of the 1000 cycles the standard asks for, and a matrix of that shape yields datasets of 535,000 symbols per direction, which is below the million the estimators need. What is reported is therefore the duplicate analysis and the sanity check, and not the row and column entropy estimates of the standard. A control arm restarting without removing power was not run on the same machine, so the distinction between the two kinds of restart is argued from the timestamp counter rather than demonstrated.

The restart test does not observe the first output after a reset. Firmware and kernel draw on the instruction before any collector runs, so the collected row begins at an unknown point in the output stream, and that point varies between boots. This is a property of the platform and bounds what the family can establish about the state of the source immediately after power is restored.

A power cycle through the management controller is not a loss of power. The machine was switched off and on again by the platform, which is not the same as removing it from its supply, and no measurement here distinguishes the two.

The floor of the assessment depends on the sample size. Every figure in this work is read against a calibration of the same length, and figures taken at different lengths are not comparable with one another. Where this chapter reports a value it reports the length beside it.

The assessment terminates only below a repetition length. Above a period between 16 and 32 kilobytes the procedure does not complete within the time allowed, and a dataset of that shape is recorded as not terminating rather than as an estimate.

Almost everything in a manifest is self-reported. Board identifiers, firmware versions and the microcode revision are read from the system under measurement, which an adversary in control of that system also controls. The manifests are sufficient for an honest collection and not against the adversary Chapter 4 assumes.

Enabling the confidential computing features changed the platform. The virtual interrupt controller was disabled as a consequence of the configuration recorded in Section 6.2.4, so interrupt handling on the host differs between the measurements taken before and after it.

The second assessment framework is not independent of the first. Two of the four tests of the AIS 31 suite are estimators of SP 800-90B, which version 3.0 adopts unchanged, and their values here come from the same assessment run. Where the two frameworks agree, they agree in part because they are looking through the same instrument.

The AIS 31 suite was implemented rather than executed. The published tool exists only as a Windows build and was used as the source of its constants and of the reference values the implementation reproduces, and it was not itself run. A result reported here is therefore a result of that implementation, validated against the published reference, and not an artefact of the tool.

That suite examines about five percent of each dataset. It reads 2,040,000 bits per attempt and at most two attempts, so a defect whose period exceeds the examined window cannot appear in its verdict, and Section 7.2 reports a dataset where this occurs.

The interception result covers RDRAND only. The control bits for the two instructions are independent, and the measurement that would have extended the result to RDSEED could not be carried out, so nothing in this work establishes whether the instruction with the full-entropy guarantee can be trapped in the same way.

The cost of an exit was measured on a client processor and through CPUID. The figure describes an ordinary transition on a mobile part rather than a trapped execution of RDRAND on a server part, and the cost of an intercepted execution of the instruction itself could not be isolated, since the instruction was absent from the CPUID of the test guest.

The floor of the assessment is not a fixed value. Runs over data of identical construction spread by more than two tenths of a bit per byte depending on which branch binds, so a figure reported here is meaningful only against a calibration of the same construction and length, and no threshold derived from one run should be applied to another.

The limitation that outlasts all of these is the one Section 7.2 establishes. No assessment of output can separate a sound source from a generator whose seed was too small, and every figure in this work is an assessment of output.

7 RESULTS

7.1 Baseline on Bare Metal

The baseline measurement asks what the two instructions deliver when nothing mediates them. Fourteen datasets were assessed, three runs per instruction on the AMD platform, three per instruction in the confidential guest reported in Section 7.7, and one per instruction on the Intel platform.

Read against the theoretical maximum of eight bits per byte, every figure falls short. Read against the calibration described in Section 6.2.6, which is what the same procedure returns for a source of known full entropy at the same length, none

TABLE VI
BASELINE MIN-ENTROPY OF BOTH INSTRUCTIONS, READ AGAINST THE CALIBRATION OF SECTION 7.2.

Environment	Instruction	Runs	min-entropy per byte
AMD bare metal	RDRAND	3	7.470486 to 7.558281
AMD bare metal	RDSEED	3	7.470487
Intel bare metal	RDRAND	1	7.470486
Intel bare metal	RDSEED	1	7.558281

of them does. The calibration runs of Section 7.2 return values between 7.408654 and 7.470488 for data of known full entropy, so the measured values sit at the upper end of that range or marginally above it. The shortfall belongs to the estimator and not to the source, which is the reason the calibration exists.

Which of the ten estimators produces the reported minimum varies from run to run. Over the fourteen datasets the binding estimator was the LZ78Y predictor six times, the MultiMCW predictor five times, the t-Tuple estimate twice and the longest repeated substring once. The Compression estimate, which the literature discussed in Section 6.1 reports as dominating the minimum for near-uniform sources in about three quarters of cases, binds in none of them. Whether the symbol-level or the bit-level branch binds also varies, eleven times the former and three times the latter, which is why Section 6.1 declines to name a branch in advance.

The compression check returns a ratio of 1.000056 for every one of the fourteen datasets, identical to six decimal places with the calibration, so none of the streams is compressible in any degree the check can see.

One difference between the platforms does not appear in the entropy figures at all. On the AMD platform both instructions delivered every value on the first attempt. On the Intel platform RDRAND did the same, while RDSEED failed on 61.6 percent of calls even in a single thread with no competing load. The stream that was assessed is the same length in both cases, because a failed call is repeated, so the failure rate is visible only in the record beside the stream. The assessment sees a retried stream and says nothing about how much

retrying produced it, which is the first of several places in this chapter where the interesting property is not the one the assessment measures.

7.2 What the Assessment Detects and What It Does Not

This section reports the ladder described in Section 6.3.2, and it is the section that bounds every other number in this chapter. Twenty-eight datasets of ten million symbols each were put through all three procedures, and the table records what each procedure saw.

Five results follow from it, and the first is the one the paper turns on.

A generator with a thirty-two bit seed escapes all three procedures. Its true min-entropy is about three millionths of a bit per byte, and the assessment returns between 7.3433 and 7.5465, which is not merely close to the calibration but overlapping it in both directions, since one of the three runs is assessed higher than any run of the calibration itself. The compression ratio agrees with the calibration to six decimal places. The AIS 31 suite passes it in all three runs, with both prediction tests between 0.994 and 0.998 against a threshold of 0.95. Three procedures from two regulatory traditions look at a stream carrying thirty-two bits of entropy and see full randomness.

The floor is a property of the branch that binds, not a threshold. The calibration runs, whose true entropy is eight bits per byte, return 7.4087, 7.4236 and 7.4705. Two of the three fall below the value the first ladder suggested as the

TABLE VII
WHAT EACH OF THE THREE PROCEDURES DETECTS ON THE LADDER OF CONSTRUCTED SOURCES.

Dataset	true H per byte	SP 800-90B	AIS 31 T_irn	xz ratio
Calibration	8	7.4087 to 7.4705	passed	1.000056
Defect, p = 0.0001	3.3219	3.7692 to 3.8105	failed	1.000056
Defect, p = 0.001	2.4914	2.9802 to 2.9875	failed	1.000056
Defect, p = 0.01	1.6610	1.9069 to 1.9105	failed	1.000056
Defect, p = 0.1	0.8305	0.8441 to 0.9272	failed	0.942112
Defect, p = 0.5	0.2500	0.2517 to 0.2667	failed	0.568306
Small seed, 32 bit	0.0000032	7.3433 to 7.5465	passed	1.000056
Repetition, period 625,000	0.5	did not terminate	passed	0.062678
Repetition, period 8,192	0.0066	0.000000	failed	0.000987
Repetition, period 2,048	0.0016	0.000000	failed	0.000366
Repetition, period 512	0.0004	0.000000	failed	0.000211
Constant	0	rejected on reading	failed	0.000158

floor, because the bit-level branch bound them. A gate placed at that value would therefore reject data of known full entropy about as often as it accepts it, which is a further reason why a measured figure is read against a calibration of the same construction and length rather than against a fixed number.

The assessment overestimates a defective source in every run. Against the rung that replaces a fraction of the words with zero while signalling success, the reported value exceeds the true min-entropy at every fraction tested, by 14.0 percent at one word in ten thousand, 19.7 percent at one in a thousand, 14.9 percent at one in a hundred, 6.1 percent at one in ten and 4.0 percent at one in two. Fifteen runs, fifteen overestimates, and the error is largest where the defect is smallest.

Each procedure has a blind spot the others do not. The compression check sees nothing in the defective rung until a tenth of the words are replaced, where the difference below that point is four bytes in ten million. The AIS 31 suite passes the repetition at period 625,000, and it does so for a structural reason rather than a statistical one, since it examines 255,000 bytes per attempt and that window lies entirely inside the first period, so no repetition occurs in what it reads. The entropy assessment does not terminate on the same dataset, while the compression check reduces it to a sixteenth of its size. The dataset that defeats two of the three procedures is trivial for the third.

Within the AIS 31 suite the two prediction tests differ sharply. On a stream repeating every 512 bytes the Multi-MMC test falls to 0.047 bits per bit while the LZ78Y test still reports 0.980, and the pattern holds at the longer periods. The suite catches short repetitions only because it requires every test to pass, and a rule that averaged its tests instead would let them through.

The matrix is the answer to the first research question. An assessment under SP 800-90B detects a source broken in the ways the ladder constructs, with one exception, and the exception is the case that matters most for a conditioned output. A generator whose seed is too small produces output that no procedure operating on output alone can distinguish from a well-seeded one. The figures reported in Section 7.1 are therefore consistent with the instructions working as specified, and they are equally consistent with a source seeded far more poorly than its specification claims. Nothing in this chapter can separate those two readings, and no measurement of output ever could.

7.3 Interception under KVM/QEMU

The interception demonstrated in 2016 reproduces on current hardware and current hypervisor code without modification to the approach. A hypervisor module was altered to enable the exit for RDRAND and to write a chosen constant into the guest's destination register with the carry flag set. A guest executing the instruction then received that constant.

Two target registers were verified, and both received the substituted value, which establishes that the handler reads the destination from the instruction information field rather than

assuming a fixed register. The value appears a second time in an independent place, namely in the register dump the guest kernel prints when its own start-up check of the instruction fails, where the constant is visible in both registers at the moment of detection.

That check is the limit of what a guest notices. The kernel draws eight values during early boot and requires at least five of them to differ from their predecessor. A substitution that returns one constant fails that test immediately. What follows from the failure has changed with the kernel version. The guest observed here ran a kernel that issued a warning and continued, whereas a current kernel clears the feature bits for both instructions and reports the source as unreliable, which removes it from the system altogether. In a confidential guest, where Section 5.5.2 shows the kernel aborting the boot when the mandated contribution cannot be obtained, the same substitution would therefore prevent the guest from starting rather than going unnoticed. That chain follows from the specifications and the kernel source and was not exercised here, for the reasons Section 6.3.3 gives.

A substitution that varies its output rather than repeating one value defeats the check, and nothing else in the guest is in a position to notice it. Section 7.2 establishes that an entropy assessment is not in that position either.

What such an interception costs the operator was measured on the client platform, using CPUID as the trapping instruction so that the transition is isolated from any property of the instruction under study. On the host the instruction takes 110 cycles once the cost of the timing is subtracted, and in a guest it takes 2,994, so the transition itself accounts for about 2,880 cycles, close to 1.1 microseconds on that part. The same run measures the instruction itself on that machine at about 1,000 cycles once the timing is subtracted, so an exit costs roughly three times what the instruction it interrupts costs, and intercepting every call would therefore not quite quadruple the price of obtaining a random number. That is expensive without being prohibitive, which bears on whether a hypervisor could sustain the interception against a guest that draws continuously rather than mount it briefly.

The cost of an intercepted RDRAND could not be measured separately. The instruction was not present in the CPUID of the test guest, so the values collected there describe something other than a trapped execution of it and are not reported. The interception itself rests on the patched module and on the specification, as above, and not on a timing measurement.

Whether RDSEED can be intercepted in the same way remains open. The measurement was designed and not carried out, for the reason Section 6.3.3 gives, and the question cannot be settled from the RDRAND result, since the two instructions are trapped through separate control bits that the specification treats as independent. The 2016 study left the same question open, and this work does not close it.

TABLE VIII
RDSEED UNDER CONTENTION ON THE AMD PLATFORM.

Threads	RDSEED delivered per second	failed calls
1	1,836,477	0.0 %
2	1,584,448	47.4 %
4	1,599,234	47.2 %
8	1,591,440	47.6 %
64	1,611,931	47.2 %

7.4 The Source under Contention

The two instructions behave differently under load, and the difference is architectural rather than incidental.

RDSEED on the AMD platform delivers about 1.6 million values per second from the second thread onward, and every additional thread produces failed calls rather than additional values. RDRAND on the same machine scales nearly linearly to the full thread count, with the per-thread rate falling from 41.9 million to 31.9 million values per second between one thread and sixty-four. The first is bounded by the physical source behind the conditioner, the second by the deterministic generator in front of it, which is the division Section 3.3.2 describes.

The placement measurement settles where that physical source sits. Four threads concentrated on a single core complex deliver 1,603,305 values per second, and four threads distributed across all eight complexes deliver 1,603,290, with failure rates of 47.2 percent in both cases. The agreement to five significant figures rules out a source per complex and leaves one source shared behind the interconnect.

The Intel platform shows the same structure at half the height. The delivered rate is pinned at about 925,000 values per second from one thread to forty-eight, while the failure rate climbs from 56.5 percent to 98.1 percent. The saturation point is therefore reached before the first additional thread, and additional demand is absorbed entirely by failures.

Under sustained memory pressure, which is the condition under which the defect on the following processor generation was reported, the AMD platform delivered 1.22 million values per second with 37.9 percent failures over 44.6 million draws, and not one call reported success while returning zero. The platform measured here is of the preceding generation, so this is a negative control that behaved as expected rather than a test of the defect itself.

7.5 Time to a Seeded Pool

The interval the BSI 2016 study measured in tens of seconds has become a property of a boot parameter.

Each figure is the mean of three real restarts on the AMD platform, twelve boots in total. Crediting the instruction shortens the exposed interval by about two orders of magnitude, and crediting the bootloader changes nothing that is visible at this resolution, which is consistent with the firmware contribution being mixed in without advancing the estimate.

The second arm shows what happens inside the longer interval. A boot with the CPU source not credited recorded 59,938 accesses to the unseeded pool before it was seeded, spread over 32 distinct callers. One caller accounts for 55,248 of them, and the remainder are spread over process creation, address space randomisation, filesystem operations and key management. None of these consumers waits, since the interfaces that block are not the ones they use.

Measured against 2016, the change is a substitution rather than a repair, as Section 4.4.2 argues. The software sources have not become faster. A source was added that makes their accumulation time irrelevant, and the interval returns in full when that source is not credited.

The Intel platform carries a reduced version of this measurement. Two boots were recorded and only the seeding timestamp of each was preserved, 0.422831 seconds in one and zero in the other. A pool seeded at a timestamp of zero is the signature of a seed carried across a restart rather than of a pool seeded instantly, and the full log that would settle which of the two occurred was not kept. No claim is made about that boot.

7.6 Behaviour Across Restarts

The restart campaign completed 535 power cycles on the AMD platform over three days, each switching the machine off and on again rather than restarting it. Every cycle produced a distinct boot identifier and collected its row within 1.166 to

TABLE IX
TIME TO A SEEDED POOL OVER TWELVE RESTARTS, THREE PER CONFIGURATION.

Configuration	Time to a seeded pool
CPU source credited	0.1327 s
CPU source not credited	7.9360 s
CPU source credited, bootloader credited	0.1332 s
CPU source not credited, bootloader credited	8.1164 s

1.223 seconds of the kernel starting, with a median of 1.184 seconds.

No row repeats. Across 535 cycles and both instructions, no row of one thousand values is identical to any other, which is the outcome a source that does not return to a repeating state produces.

The sanity check of the standard passes in both directions. With the bound simulated separately for rows of one thousand values and for columns of 535, as Section 6.3.6 requires, a row may hold at most 22 occurrences of one symbol and a column at most 16. The largest count observed in any row is 16 for RDRAND and 15 for RDSEED, and the largest in any column is 12 for RDRAND and 10 for RDSEED.

The bulk samples collected behind the rows answer a further question, which is what the source delivers immediately after power is restored. Across five cycles spread over the campaign the assessment returns between 6.655551 and 7.498 bits per byte for both instructions. Those figures are lower than the ones in Section 7.1 and they mean the same thing, because the datasets are ten times smaller and the floor of the procedure is correspondingly lower and wider. Three calibration runs at exactly that length return 6.655551, 7.166327 and 7.238458, and the lowest measured value agrees with the lowest calibration value to six decimal places. Output taken immediately after a cold start is therefore indistinguishable from full entropy by the same standard that Section 7.2 shows cannot distinguish either of them from a badly seeded generator.

What an intercepted source would look like in this test is visible in the pilot of the campaign, where the interception was active. The collected row holds two distinct byte values, the more frequent of which occurs 999 times out of 1000, against a bound of 22, and the same row appears identically in two different boots. Both properties the test examines fail at once, and they fail by margins that require no estimate to see.

7.7 Measurements under Confidential Computing

Three runs per instruction were collected inside a SEV-SNP guest on the AMD platform, and the entropy figures are reported in Section 7.1 alongside the others. They are the same figures. RDRAND in the guest returns between 7.470486 and 7.568351 bits per byte, RDSEED returns 7.470486 to 7.470487, every call succeeded, and the compression check returns the same ratio as everywhere else. By every measure this study applies, the source inside the confidential guest is indistinguishable from the same source on the same machine outside it.

That is the empirical content of the confidential computing part of this work, and it is narrower than it may appear. It establishes that the isolation does not degrade what the instruction delivers. It cannot establish what the instruction delivers, for the reason Section 7.2 gives.

Three further questions about this environment were not answered by measurement. Whether a Trust Domain behaves the same way could not be tested, since the Intel platform exposed no way to create one, and Section 5.2.1 therefore

rests on the specification and the published module source. Whether the instruction can be intercepted on AMD needs no test, since neither the architecture manual nor the kernel source defines an intercept to enable. Whether the host can withdraw the instruction through the CPUID surface of the guest was designed as a measurement and not carried out, and Section 5.6 rests on the firmware interface instead.

One negative result belongs here rather than among the numbers. No attestation report accompanies any of the guest datasets, because the kernel of the rented environment carried no driver through which one could be requested. The claim that these measurements come from a confidential guest therefore rests on what the guest kernel reported about itself and on the launch configuration as the operator recorded it afterwards, and not on a signed statement. The environment that the architecture makes verifiable is, in this study, the only one whose provenance could not be verified.

A difference in elapsed time between the guest and bare metal was observed and is not reported as a result. The collection tool times the generating loop together with the hashing and writing of the collected data, the guest wrote through a virtual device into an image file while bare metal wrote to local storage, and the runs are short enough that start-up costs are of the same order as the difference itself. The control measurement that would separate these was not possible after the platform was released.

7.8 Measured Output Against Vendor Certification

The certificates both vendors hold cover the raw output of the noise source before conditioning, as Section 2.3.1 sets out. What this study measures is what the instructions return, which is output that has passed through conditioning and, for RDRAND, through a deterministic generator as well. The two quantities describe different points in the same pipeline, and the figures of one are not the figures of the other.

What can be said from the measurements is bounded accordingly. The output of both instructions on both platforms is assessed at the floor of the procedure, which is what a full-entropy source produces. Nothing in that result reaches the noise source, and nothing in the certificates reaches the output. The relationship between them is an assumption about the conditioning stage, supplied by the vendor and examined by a laboratory under agreement with the vendor.

That is the position a relying party occupies, and it is the same position for both vendors although their disclosure differs in shape, as Section 3.4.2 describes. Chapter 8 takes up what follows from it.

8 DISCUSSION

The four questions of Section 1.3 are answered by different parts of this work, and the answers are collected here before the discussion takes them further.

RQ1, what an assessment can establish. Section 7.2 answers it directly. An assessment under SP 800-90B detects a source into which a defect has been injected, at every injected fraction tested, and it overestimates the entropy of such a source in every run rather than underestimating it. Against a generator seeded with too little entropy it establishes nothing at all, since that stream receives the same figure as the real instruction output to six decimal places. The assessment is therefore evidence about the output and not about the source.

RQ2, whether interception remains feasible. Section 7.3 answers it affirmatively for Intel and by specification for AMD. The attack reproduces on current hypervisor code, it requires a handler in addition to the two control bits because the default path returns an invalid opcode, and it has no counterpart on AMD, where neither the manual nor the kernel source defines an intercept to enable. The behaviour therefore differs between the vendors in the way the 2016 study inferred.

RQ3, what confidential computing removes. Sections 5.2.1, 5.3.2 and 7.7 answer it jointly. Both architectures remove the instruction intercept, on Intel by placing the execution controls of a Trust Domain outside the host's reach and on AMD by there being nothing to remove. Measurement inside a SEV-SNP guest finds the source indistinguishable from the same source outside it. What remains with the host is the CPUID surface through which the instruction is announced, and what the entropy supply of a confidential guest rests on is the proposition that a single unverifiable source delivers what its specification claims.

RQ4, the conditions for sufficiency. Sections 8.1 to 8.4 answer it, and the answer is a set of conditions rather than a verdict. The instruction is the narrower trust assumption than any alternative reachable from inside a guest, the combiner no longer compensates for its failure, and no deployment subject to the physical classes of the German scheme can satisfy them with the instruction alone.

8.1 Sufficiency of RDRAND Without an External HSM

The question that motivated this work asks whether a cloud deployment can rely on the CPU instructions instead of a hardware security module, and the results permit an answer with a precise shape rather than a verdict.

What the measurements establish is that the output of both instructions, on both vendors, on bare metal and inside a confidential guest, is indistinguishable from full entropy by the strongest procedure the field has agreed on. Fourteen datasets sit on the floor of that procedure, no compression check finds structure in any of them, and the source delivers the same output immediately after a cold start as it does in steady state. If the question is whether anything in the delivered output suggests a problem, the answer is that nothing does.

What the measurements cannot establish is the proposition a relying party actually needs, which is that the output carries the entropy its specification claims. Section 7.2 shows a stream with thirty-two bits of total entropy receiving the same assess-

ment as the real instruction output, to six decimal places. Any argument of the form that the output was measured and found sound therefore proves less than it appears to, and this applies to every published assessment of a conditioned source, not only to this one.

The comparison with a hardware security module has to be made on that footing. A module of the kind Section 2.3.1 describes is not trusted because its output was measured. It is trusted because an accredited laboratory examined the noise source, the conditioning and the key storage as a physical assembly, because the device carries a protection perimeter that reacts to being opened, and because the evidence of that examination is public. None of those three properties is available for a CPU instruction, and no amount of output measurement substitutes for them.

The practical situation is nevertheless not a choice between a verified and an unverified source. Reaching a physical module from inside a virtual machine requires either mediation by the hypervisor or a provider interface, and both re-introduce the party that confidential computing was adopted to exclude. A tenant choosing between the instruction and a provider-mediated module is choosing between trusting the CPU vendor and trusting the CPU vendor together with the provider. On that comparison the instruction is the narrower assumption, which is the sense in which it is sufficient, and it is a different sense from the one the question is usually asked in.

8.2 The Combiner as a Mitigation

The Linux combiner is the strongest structural argument available for using a source that cannot be verified, and confidential computing removes most of its force.

The guarantee is stated in Section 2.2.3. A combiner over several sources produces output that remains sound as long as one contribution carries entropy the adversary cannot predict and is independent of the others. The guarantee costs nothing when it holds, and it holds under an adversary who controls some sources but not all of them.

Under confidential computing the set of sources the guarantee could draw on collapses. Section 5.5.1 assesses eleven sources and finds that the hypervisor mediates, degrades or supplies all of them except the CPU instruction. The combiner still runs and still mixes, but the proposition it needs, that one independent contribution is beyond the adversary, is now a proposition about exactly one source. A construction that tolerated the failure of any single source has become one that tolerates the failure of none.

Three properties of the combiner survive this and are worth stating, because the collapse is often described as total. The combiner still protects against sources that are weak rather than adversarial, which is the ordinary case. It still ensures that a contribution which does carry entropy is not diluted by contributions that do not. And it still defends against a hypervisor that is honest but degraded, which is the threat model of the 2016 study and remains the common case in practice.

What it no longer does is protect against the case this paper is about. Against a hypervisor that is adversarial and a CPU vendor that is trusted, the combiner is unnecessary, since the instruction is already outside the adversary's reach. Against a CPU vendor that is not trusted, the combiner has nothing left to mix in. The argument that carried the decision described in Section 3.3.3, that the instruction would be used but never alone, has quietly become an argument that it is used alone.

8.3 Conditions for Trusting RDRAND and RDSEED in Cloud Deployments

The results support a set of conditions rather than a recommendation for or against. Each of them follows from a specific finding in Chapter 7.

Credit the CPU source at boot. Section 7.5 measures the cost of withholding that credit as two orders of magnitude in the length of the interval during which the pool is not seeded, and it records 59,938 accesses to the unseeded pool during that interval from consumers that do not wait. Withholding the credit out of distrust for the vendor exchanges a trust risk for an availability risk and leaves the same vendor in the path either way.

Prefer RDSEED where the design allows it. The two instructions differ in what they guarantee across successive outputs, as Section 3.3.2 sets out, and Section 7.4 shows the price of that difference to be throughput rather than reliability on one of the two platforms. Where a system seeds a generator rather than consuming random numbers continuously, the slower instruction is the one whose guarantee matches the use.

Record the failure rate, not only the output. Section 7.1 reports a platform where a majority of calls to one instruction fails under no load at all, a fact invisible in the collected stream because failed calls are repeated. A deployment that monitors only the values it receives cannot see the source degrading until it stops delivering entirely.

Require an attestable image, not only an attestable platform. Section 7.7 reports a confidential guest whose provenance could not be established, because the image the provider supplied carried no driver through which a report could be requested. The architectural guarantee exists and the tenant could not reach it. This belongs in procurement rather than in code.

Treat the CPUID surface as part of the attack surface. Section 5.6 shows that the host still supplies the guest's view of which instructions exist, and that withdrawing an instruction on that path is an established operation rather than a hypothetical one. A guest that verifies the presence of the instruction it depends on notices this, and a guest that assumes it does not.

Do not treat an output assessment as verification. Section 7.2 is the reason. An assessment answers whether the output shows a defect of a kind the procedure can see, and a deployment that reports a passing assessment as evidence of entropy has overstated what it did.

8.4 Implications for BSI High-Security Requirements

The German scheme takes a position that the results of this work make concrete rather than abstract, because it asks for evidence about the noise source rather than about the output.

Section 2.3.4 describes the structure. The physical classes of AIS 20/31 require a stochastic model of the noise source, which is a mathematical description of the physical process together with an argument for why it produces the entropy claimed. A model of that kind cannot be constructed from output, and it cannot be constructed by a party without access to the raw noise. For a CPU instruction on a rented machine, both are out of reach, and the results here show exactly how far out of reach. The output looks the same whether the source behind it is sound or not.

This has a consequence for a deployment that must satisfy the physical classes, which is that it cannot satisfy them with the instruction alone, whatever the output assessment says. The vendor holds the evidence that would be needed and does not publish it in a form the scheme accepts, and the instruction carries no protection perimeter of the kind the corresponding module classes require.

For deployments below that bar, the conditions of Section 8.3 apply and the instruction is defensible. The distinction between the two cases is not the quality of the output, which this work measured and found unremarkable in the best sense, but the kind of evidence the deployment is required to produce.

One further consequence deserves stating for the scheme itself rather than for its users. The assessment procedure that both schemes rely on for output, and that this paper applied in its reference implementation, does not distinguish a sound source from a badly seeded generator, and it overestimates the entropy of a source with an injected defect in every run of the ladder in Section 7.2. A framework that accepts such an assessment as evidence about a conditioned source is accepting less than it appears to, and the ladder is reproducible by anyone who wishes to check that claim.

9 CONCLUSION

9.1 Summary of Findings

This work set out to revisit a study from 2016 on current hardware and to carry its questions into an architecture that did not exist when it was written. Four findings carry the result.

The output of RDRAND and RDSEED cannot be distinguished from full entropy by the procedures available to a party outside the manufacturer. Fourteen datasets across two vendors, bare metal and a confidential guest, all sit on the floor of the assessment, and the same holds for output collected immediately after a cold start across 535 power cycles.

That statement is worth less than it appears, and this is the central result rather than a caveat. A generator seeded with

thirty-two bits of entropy receives the same assessment, to six decimal places, as the real instruction output. The procedure that both regulatory schemes rely on for conditioned output cannot separate the two, and neither can a compression check. An assessment of output is evidence about the output and not about the source behind it.

The interception demonstrated in 2016 remains feasible on current hypervisor code, and the architecture that removes it does so only on the instruction path. Two target registers were verified under a modified hypervisor, and the guest observed no difference beyond what its own start-up check catches. Under confidential computing the intercept is gone, and the host retains its influence over what the guest is told about the instruction through the interface that announces it.

The combiner that made the use of an unverifiable source defensible does not survive the transition. Of eleven sources feeding the pool, confidential computing changes the assessment of exactly one, and the guarantee that required one independent uncompromised contribution now rests on that one.

9.2 Comparison to BSI 2016

Measured against the study this paper revisits, three of its findings have changed and one has not.

The boot-time interval it measured in tens of seconds is now a property of a boot parameter. With the CPU source credited the pool is seeded after 0.1327 seconds, and without it after 7.9360 seconds on the same machine. The software sources did not become faster. A source was added whose accumulation time is negligible, and the interval returns in full when that source is not credited.

The interception it demonstrated still works. What has changed is the cost of mounting it, since the hypervisor of today routes the exit to an invalid opcode by default, so an attacker must supply a handler as well as enable the bit. The ten lines the study reported are still roughly the right number.

The asymmetry it observed between the two vendors persists. Neither the architecture manual nor the kernel source defines an intercept for either instruction on AMD, and the interception it demonstrated on Intel therefore has no counterpart there.

What has not changed is the position of the relying party. The 2016 study assumed a non-malicious hypervisor and took the entropy of the instruction from design documentation. This work removes the first assumption and finds that the second cannot be removed at all.

9.3 Recommendations for Practitioners

The conditions in Section 8.3 are the operational form of this work, and three of them deserve repeating as the ones most often got wrong.

Credit the CPU source at boot. Withholding the credit is a common reflex for operators who distrust the vendor, and it does not remove the vendor from the path. It lengthens the

interval during which the pool is unseeded and leaves tens of thousands of consumers drawing from it.

Require an image that can attest. The architecture provides the one mechanism by which a tenant could verify its environment, and whether a tenant can use it is decided by the image the provider supplies. This is a procurement condition, and the measurements in this work failed it.

Do not report a passing entropy assessment as evidence of entropy. It is evidence that the output shows no defect of a kind the procedure can see, and Section 7.2 sets out how much that excludes.

9.4 Future Work

Three directions follow from what this work could not do.

The first is a measurement inside a Trust Domain. Everything this paper says about TDX rests on the specification and on published source, and the corresponding statements about SEV-SNP rest on measurement. Closing that gap requires a platform that exposes Trust Domain creation to the tenant, which is a matter of availability rather than of method.

The second is the path that survives the intercept. The host still supplies the guest's view of which instructions exist, and a guest that is told an instruction is absent does not call it. Measuring how far that reaches, and what a guest can do to notice it, is a small experiment that this work designed and did not carry out.

The third is the one that matters most and is the hardest to reach. Every conclusion in this paper is bounded by the fact that a conditioned output carries no evidence about the source behind it. Changing that requires either access to the raw noise, which the vendors do not provide, or a construction in which the guest does not have to rely on a single source it cannot inspect. The second is the more promising direction, and the service module work described in Section 5.6 suggests the architecture may move that way for reasons unrelated to entropy.

APPENDIX A: CHECKLIST FOR RDRAND ASSESSMENT IN CLOUD ENVIRONMENTS

The checklist collects what this work found to be worth establishing before a deployment relies on the CPU instructions, in the order in which the questions arise. Each item names the section that supports it. An item that cannot be answered is not a failure of the checklist but the answer to the question of what the deployment can verify.

Before the deployment

- Does the image the provider supplies carry the driver through which an attestation report is requested? Without it the platform guarantee exists and the tenant cannot reach it (Section 7.7).
- Does the rental grant firmware access, and if not, which properties of the platform become unverifiable as a result (Section 6.2.4)?
- Which processor generation will the workload run on, recorded as family, model, stepping and microcode revision rather than as a product name (Section 6.2.5)?
- Is that generation affected by a known defect of the instructions, and does the recorded microcode revision carry the mitigation (Section 3.4.1)?

At boot

- Is the CPU source credited, and is that decision deliberate rather than inherited from a template (Section 7.5)?
- Does the guest verify that the instruction it depends on is announced as present, rather than assuming it (Section 5.6)?
- Does the kernel report its start-up check of the instruction as passed (Section 7.3)?
- For a confidential guest, is the attestation report requested, stored and checked against an expected platform state (Section 5.4)?

During operation

- Is the rate of failed calls recorded alongside the values received, rather than only the values (Section 7.1)?
- Is the seeding state of the pool observed, so that a delay becomes visible rather than silent (Section 7.5)?
- Are the consumers that draw before the pool is seeded known for the images in use (Section 7.5)?

When assessing the source

- Is the assessment run against a calibration of the same sample size, so that the floor of the procedure is known (Section 6.2.6)?
- Is the sample size fixed and reported with every figure (Section 6.1)?
- Is a known-defective source assessed alongside the real one, so that what the procedure detects is established rather than assumed (Section 7.2)?
- Is the result reported as the absence of a detectable defect rather than as evidence of entropy (Section 8.3)?

What the checklist cannot establish

- Whether the noise source behind the instruction carries the entropy its specification claims. No procedure operating on output can answer this, and Section 7.2 shows what it costs to assume otherwise.

APPENDIX B: TEST ENVIRONMENT SPECIFICATIONS

Every dataset carries a manifest recording the machine it was collected on, written at collection time by reading the running system. The table below gives one manifest per environment; the fields are identical across the datasets of an environment unless noted. Section 6.2.5 sets out what these records can and cannot establish.

TABLE X
MEASUREMENT ENVIRONMENTS AS RECORDED IN THE MANIFESTS.

Environment	Board	Processor	Fam/Mod/ Step	Sockets x Cores x Threads	L3	Kernel	Microcode
AMD EPYC 9354, bare metal	Supermicro H13SSL-N	AMD EPYC 9354 32-Core Processor	25 / 17 / 1	1 x 32 x 2	256 MiB (8 instances)	7.0.0-31-generic	0xa101158
AMD EPYC 9354, SEV-SNP guest	Supermicro H13SSL-N	AMD EPYC 9354 32-Core Processor	25 / 17 / 1	1 x 32 x 2	256 MiB (8 instances)	7.0.0-31-generic	0xa101158
Intel Xeon 6527P, bare metal	Supermicro X14SBT-G	Intel(R) Xeon(R) 6527P	6 / 173 / 1	1 x 24 x 2	144 MiB (1 instance)	5.15.0-177-generic	0x1000405
Intel Core i7-1255U, KVM guest	Pilot Standard PC (Q35 + ICH9, 2009)	12th Gen Intel(R) Core(TM) i7-1255U	6 / 154 / 4	4 x 1 x 1	64 MiB (4 instances)	7.0.0-31-generic	0x43b

Three properties of these environments are not visible in the table and bear on how the results are read.

The AMD platform ran with SEV, SEV-ES and SEV-SNP enabled in the host, reporting platform security firmware SEV API:1.58 build:2. The firmware settings that had to be changed to reach that state are recorded in Section 6.2.4.

The Intel machine was measured in two software states. The baseline and contention datasets were collected from a volatile RAM-resident system on kernel 5.15.0-177-generic with an earlier build of the collection tool, whereas the interception work and the boot-time measurement ran on the installed system on kernel 7.0.0-31-generic with the TDX host initialised. Rates measured on the two states are not interchangeable.

The guest row describes what the guest kernel reported about itself. No attestation report accompanies the confidential guest datasets, and the manifests stored with them were written on the host, so they carry host rather than guest context. Sections 6.2.5 and 6.5 set out what follows from that.

APPENDIX C: RAW MEASUREMENT DATA

The streams assessed in this work are listed below with the parameters they were collected under and the digest of the bytes themselves, so that a reported figure can be traced to the exact data it came from. The digest is computed by the collection tool while the stream is written and repeated in the manifest. The data are held outside the manuscript repository, in the collection that accompanies this report and that carries the analysis results, the metadata and the tooling alongside the raw streams. A stream that can be rebuilt from a recorded seed, or checked against the digest below, is described there rather than stored twice. Where a dataset was collected for its rate and its failure count rather than for its content, the stream itself was not kept, and the table says so in place of a digest that nothing could be checked against.

TABLE XI
COLLECTED DATASETS WITH THEIR PARAMETERS AND DIGESTS.

Dataset	Instruction	Threads	Symbols	Failure rate	SHA-256 (first 16)
AMD EPYC 9354, bare metal					
m1-bm-rdrand-run02	rdrand	1	10 000 000	0.000 %	fc61bc485a1d916f
m1-bm-rdrand-run03	rdrand	1	10 000 000	0.000 %	43de6538a6ec9794
m1-bm-rdseed-run02	rdseed	1	10 000 000	0.000 %	7a72dc0e5973568d
m1-bm-rdseed-run03	rdseed	1	10 000 000	0.000 %	778a2d127091a199
AMD EPYC 9354, SEV-SNP guest					
m1-snp-rdrand-run01	rdrand	1	10 000 000	0.000 %	58ca1a3145dfa0d7
m1-snp-rdrand-run02	rdrand	1	10 000 000	0.000 %	9a5f7e48641cd10d
m1-snp-rdrand-run03	rdrand	1	10 000 000	0.000 %	395345a2e28054a6
m1-snp-rdseed-run01	rdseed	1	10 000 000	0.000 %	540259a666d054f0
m1-snp-rdseed-run02	rdseed	1	10 000 000	0.000 %	8b3af34bf886bd46

Dataset	Instruction	Threads	Symbols	Failure rate	SHA-256 (first 16)
m1-snp-rdseed-run03	rdseed	1	10 000 000	0.000 %	c6fc5be7f2f90a98
AMD EPYC 9354, contention series					
m4-baseline-rdrand	rdrand	1	10 000 000	0.000 %	7560f257c724b69b
m4-baseline-rdseed	rdseed	1	10 000 000	0.000 %	d0d77a71e8fa69b9
m4-ccd-conc-1	rdseed	4	10 000 000	47.246 %	n/a
m4-ccd-conc-2	rdseed	4	10 000 000	47.249 %	n/a
m4-ccd-conc-3	rdseed	4	10 000 000	47.242 %	n/a
m4-ccd-concentrated	rdseed	4	2 000 000	47.261 %	n/a
m4-ccd-dist-1	rdseed	4	10 000 000	47.238 %	n/a
m4-ccd-dist-2	rdseed	4	10 000 000	47.244 %	n/a
m4-ccd-dist-3	rdseed	4	10 000 000	47.242 %	n/a
m4-ccd-distributed	rdseed	4	2 000 000	47.246 %	n/a
m4-ccd8-eight	rdseed	8	10 000 000	47.628 %	n/a
m4-ccd8-one	rdseed	8	10 000 000	47.633 %	n/a
m4-load-mem	rdseed	64	10 000 000	41.952 %	n/a
m4-load-mem-draw	rdseed	64	10 000 000	37.927 %	n/a
m4-rdrand-long-1	rdrand	1	200 000 000	0.000 %	not retained
m4-rdrand-long-16	rdrand	16	200 000 000	0.000 %	n/a
m4-rdrand-long-2	rdrand	2	200 000 000	0.000 %	n/a
m4-rdrand-long-32	rdrand	32	200 000 000	0.000 %	n/a
m4-rdrand-long-4	rdrand	4	200 000 000	0.000 %	n/a
m4-rdrand-long-64	rdrand	64	200 000 000	0.000 %	n/a
m4-rdrand-long-8	rdrand	8	200 000 000	0.000 %	n/a
m4-rdrand-scale-1	rdrand	1	20 000 000	0.000 %	not retained
m4-rdrand-scale-16	rdrand	16	20 000 000	0.000 %	n/a
m4-rdrand-scale-64	rdrand	64	20 000 000	0.000 %	n/a
m4-scale-1	rdseed	1	2 000 000	0.000 %	not retained
m4-scale-16	rdseed	16	2 000 000	30.243 %	n/a
m4-scale-2	rdseed	2	2 000 000	47.354 %	n/a
m4-scale-32	rdseed	32	2 000 000	36.311 %	n/a
m4-scale-4	rdseed	4	2 000 000	47.229 %	n/a
m4-scale-48	rdseed	48	2 000 000	42.228 %	n/a
m4-scale-64	rdseed	64	2 000 000	47.235 %	n/a
m4-scale-8	rdseed	8	2 000 000	47.619 %	n/a
Intel Xeon 6527P, bare metal					
m1-rdrand	rdrand	1	10 000 000	0.000 %	fabc9bb55630b638
m1-rdseed	rdseed	1	10 000 000	61.646 %	e86d99bbac1b0798
Intel Xeon 6527P, contention series					
m4-16t	rdseed	16	2 000 000	96.751 %	n/a
m4-1t	rdseed	1	2 000 000	56.550 %	a72e426f9c78c4b0
m4-2t	rdseed	2	2 000 000	79.515 %	n/a
m4-32t	rdseed	32	2 000 000	98.123 %	n/a
m4-48t	rdseed	48	2 000 000	98.134 %	n/a
m4-4t	rdseed	4	2 000 000	88.813 %	n/a
m4-8t	rdseed	8	2 000 000	93.941 %	n/a

The datasets of the detection ladder are constructed rather than collected, so each is listed with the min-entropy it carries by construction and with the seed from which the stream can be rebuilt. A dataset whose seed is absent is a constant stream, which needs none.

TABLE XII
CONSTRUCTED DATASETS OF THE DETECTION LADDER.

Dataset	True H per byte	SHA-256 (first 16)	Seed (first 16)
cal-run01	8.0	9334bcc4efcd1467	d1493a35614a119d
cal-run02	8.0	079d43b4fb9b38d0	65c4c3a43044f162
cal-run03	8.0	7b9cf1c22f271c33	b7cac244602009e5
d1-p0.0001-run01	3.321927255213278	2ec768391e5595f1	2e64a70639ca35f0
d1-p0.0001-run02	3.321927255213278	92a33335e54d23be	32fe849805db4ce2
d1-p0.0001-run03	3.321927255213278	f23aead11c5cccc1	491eb8d0ac2cd2ef
d1-p0.001-run01	2.491445987273604	c041e3aa98689680	c20c77249f6df589
d1-p0.001-run02	2.491445987273604	9bb1648080506f83	25fb3b886b1b0f80
d1-p0.001-run03	2.491445987273604	daf410b3bfb6956d	c281f5083f4c3ca7
d1-p0.01-run01	1.6609640391300668	4ee99c8915589bbf	2b61a5d2f3ae27ca
d1-p0.01-run02	1.6609640391300668	9a3d07b337aefc7c	45c9986f3d9dd837
d1-p0.01-run03	1.6609640391300668	d5c38a3fcae4766d	cf5fe799d1437ffb
d1-p0.1-run01	0.8304820229660574	a051a8629cffabe4	efbe111a8a996080
d1-p0.1-run02	0.8304820229660574	7d51b011473e9baf	327fcd8454778549
d1-p0.1-run03	0.8304820229660574	263df143de896e5e	0e893ed11f4f727a
d1-p0.5-run01	0.2499999999160241	c4b37c0720f7e408	fb19ab8358270daa
d1-p0.5-run02	0.2499999999160241	10abeaa4019b22f7	794b6d13e87da3b3
d1-p0.5-run03	0.2499999999160241	65b032cc5e07ed6b	d1052ca067714d65
d2-32bit-run01	3.2e-06	a2b024bb8f9e2764	6ae53d592496041f
d2-32bit-run02	3.2e-06	8377742c7bd5b69c	0372d9d76258f4e3
d2-32bit-run03	3.2e-06	8c41f2a5c7d537df	e7deb3156fb431fa
d3-run01	0.5	898c6120f437a69a	dadd59d1e9f196bf
d3-p512	0.0004096	aeb876092062d680	b3499c502d92f052
d3-p2048	0.0016384	786470b90eb12265	e84abefe8ed4f0c7
d3-p8192	0.0065536	db475bff0f3d13d7	026e4cff2ad7cdfc
d4-run01	0.0	f5e02aa71e67f41d	n/a
d4-run02	0.0	f5e02aa71e67f41d	n/a
d4-run03	0.0	f5e02aa71e67f41d	n/a

All datasets were written by the same collection tool. The Intel baseline and contention datasets predate the version control of the tool and carry no commit identifier, as Appendix B notes. Every stream holds delivered values only, since a failed instruction contributes nothing and the call is repeated, so a failure rate above zero describes the behaviour of the source and not a gap in the data.

The restart campaign is not listed above, because its datasets are matrices rather than streams. It comprises 535 complete power cycles, each contributing one row of 1000 values per instruction together with a bulk sample of 1 048 576 values, and the per-cycle records carry the boot identifier, the timing of the collection relative to boot and the digest of each row.

APPENDIX D: REFERENCES

- [1] L. Bello, “Predictable Random Number Generator in the Debian OpenSSL Package”, Debian Security Advisory DSA-1571, 2008. <https://www.debian.org/security/2008/dsa-1571>
- [2] M. Bendel, “Hackers Describe PS3 Security As Epic Fail, Gain Unrestricted Access”, *Exophase*, 2010.
- [3] BSI, *Analysis of Random Number Generation in Virtual Environments*, 2016. <https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/ZufallinVMS/Randomness-in-VMs.pdf>
- [4] D. J. Bernstein, T. Lange, R. Niederhagen, “Dual EC: A Standardized Back Door”, 2015. <https://projectbullrun.org/dual-ec/documents/dual-ec-20150731.pdf>
- [5] Intel, *Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification*, document 348549-008US, May 2026. Section 11.7 (Instructions Restrictions) enumerates the instructions a Trust Domain may not execute and those reflected to the

- guest; RDRAND and RDSEED appear in neither list. <https://cdrdv2-public.intel.com/867568/intel-tdx-module-base-spec-348549008.pdf>
- [6] Intel, *TDX Module source*, branch tdx_1.5, module version 1.5.34. include/auto_gen_1_5/td_vmcs_fields_lookup.h sets PROCBASED_CTLDS2_INIT with the RDRAND and RDSEED exiting bits cleared and outside the mask of variable bits. <https://github.com/intel/tdx-module>
 - [7] A. Kleen, E. Reshetova, *Intel Trust Domain Extension Linux Guest Kernel Security Specification*, Intel open source documentation, 2023. <https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html>
 - [8] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, D. Wichs, “Security Analysis of Pseudo-Random Number Generators with Input”, *ACM CCS 2013*. <https://eprint.iacr.org/2013/338.pdf>
 - [9] C. E. Shannon, “A Mathematical Theory of Communication”, *Bell System Technical Journal*, 1948. <https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf>
 - [10] L. Reyzin, “Some Notions of Entropy for Cryptography”, *ICITS 2011*. <https://www.cs.bu.edu/~reyzin/papers/entropy-survey.pdf>
 - [11] NIST, *SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation*, 2018. <https://csrc.nist.gov/pubs/sp/800/90/b/final>
 - [12] NIST CMVP, *Entropy Certificate E65: Intel Digital Random Number Generator*, 2018. https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/entropy/E65_PublicUse.pdf
 - [13] NIST, *SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, 2015. <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf>
 - [14] Intel, *Intel Digital Random Number Generator Software Implementation Guide*, Section 3.2, Component Architecture. <https://www.intel.com/content/www/us/en/content-details/864722/intel-digital-random-number-generator-software-implementation-guide.html>
 - [15] A. C. Yao, “Theory and Applications of Trapdoor Functions”, *FOCS 1982*.
 - [16] D. Boneh, V. Shoup, *A Graduate Course in Applied Cryptography*, Chapter 3.5. <https://toc.cryptobook.us/>
 - [17] Intel, *TDX Linux Guest Kernel Security Specification*, which classifies RDRAND and RDSEED as outside host and VMM control. <https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html>
 - [18] NIST, *FIPS 140-3: Security Requirements for Cryptographic Modules*, 2019. <https://csrc.nist.gov/pubs/fips/140-3/final>
 - [19] NIST and Canadian Centre for Cyber Security, *Cryptographic Module Validation Program*. <https://csrc.nist.gov/projects/cryptographic-module-validation-program>
 - [20] NIST, *Entropy Source Validation Program*. <https://csrc.nist.gov/projects/cryptographic-module-validation-program/entropy-validations>
 - [21] NIST CMVP, *Entropy Certificate E179: Intel IP-DRNG 4*, 2023.
 - [22] BSI, *A Proposal for Functionality Classes for Random Number Generators*, AIS 20/31 Version 3.0, September 2024. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Certification/Interpretations/AIS_31_Functionality_classes_for_random_number_generators_e_2024.pdf
 - [23] E. Barker, J. Kelsey, K. McKay, J. Mittmann, M. Peter, W. Schindler, M. Sönmez Turan, *NIST IR 8446: Bridging the Gap Between Standards on Random Number Generation, Comparison of SP 800-90 Series and AIS 20/31*, final version, January 2026. <https://doi.org/10.6028/NIST.IR.8446>
 - [24] S. Müller, R. Romero, *Documentation and Analysis of the Linux Random Number Generator*, Version 6.4, BSI / atsec information security GmbH, December 2025, covering Linux kernel 6.18. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN_V6_4.pdf
 - [25] J. A. Donenfeld, “Random Number Generator Rewrite, Linux 5.17 and 5.18”, 2022. <https://www.zx2c4.com/projects/linux-rng-5.17-5.18/>
 - [26] B. Barak, R. Shaltiel, E. Tromer, “True Random Number Generators Secure in a Changing Environment”, *CHES 2003*.
 - [27] T. Ristenpart, S. Yilek, “When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography”, *NDSS 2010*, Internet Society. <https://rist.tech.cornell.edu/papers/ndss10x.pdf>
 - [28] NIST, *SP 800-90C: Recommendation for Random Bit Generator Constructions*, draft.
 - [29] Cryptography Research Inc., *Analysis of Intel’s Ivy Bridge Digital Random Number Generator*, 2012.
 - [30] G. T. Becker, F. Regazzoni, C. Paar, W. P. Burleson, “Stealthy Dopant-Level Hardware Trojans”, *CHES 2013*.
 - [31] “Remove RdRand from /dev/random”, public petition and the ensuing discussion on the Linux kernel mailing list, 2013.
 - [32] AMD, *AMD RNG ESV Public Use Document*, document version 0.7, 24 October 2024, NIST CMVP entropy certificate E27. Table 3 gives min-entropy per 128-bit RDSEED output by processor family and Tables 4 to 6 give the number of blocks required per security strength. https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/entropy/E27_PublicUse.pdf
 - [33] AMD, *AMD64 Architecture Programmer’s Manual, Volume 3: General-Purpose and System Instructions*, entries for RDRAND and RDSEED.
 - [34] AMD, *RDSEED Failure on AMD “Zen 5” Processors*, security bulletin AMD-SB-7055, CVE-2025-62626, CVSS 7.2. <https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7055.html>
 - [35] A. Honig, N. Porter, *7 Ways We Harden Our KVM Hypervisor at Google Cloud: Security in Plaintext*, Google Cloud blog, 2017. <https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext>
 - [36] Amazon Web Services, *The Security Design of the AWS Nitro System*, AWS whitepaper. <https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/>
 - [37] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C: System Programming Guide*. Chapter 24 (Virtual Machine Control Structures) and Chapter 25 (VMX Non-Root Operation). Section numbers cited follow current revisions, in which the Secondary Processor-Based VM-Execution Controls appear as Section 25.6.2 and Table 25-7, numbered 24.6.2 and Table 24-7 before the chapter renumbering. The format of the VM-exit instruction-information field for RDRAND and RDSEED is given in Table 30-12 of order number 325384-092US, June 2026.
 - [38] AMD, *AMD64 Architecture Programmer’s Manual, Volume 2: System Programming*, Publication No. 24593, Rev. 3.45, July 2026. Chapter 15 (Secure Virtual Machine), Section 15.9 (Instruction Intercepts) with Table 15-7, Section 15.10 (Decode Assists), Section 15.36.23 (CPUID handling under SEV-SNP), Appendix B (Layout of VMCB) and Appendix C (SVM Intercept Exit

- Codes). Neither RDRAND nor RDSEED appears anywhere in the volume. https://docs.amd.com/v/u/en-US/24593_3.45_APM_Vol2_PUB
- [39] vLibOS: Babysitting OS Evolution with a Virtualized Library OS, arXiv:1801.07880, 2018, Table 2 (VM entry 531 cycles, VM exit 481 cycles).
 - [40] G. J. Popek, R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures”, *Communications of the ACM*, Vol. 17, No. 7, 1974.
 - [41] J. S. Robin, C. E. Irvine, “Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”, *Proceedings of the 9th USENIX Security Symposium*, 2000.
 - [42] K. Adams, O. Agesen, “A Comparison of Software and Hardware Techniques for x86 Virtualization”, *Proceedings of ASPLOS XII*, 2006.
 - [43] Linux kernel source, arch/x86/kernel/cpu/rdrand.c, function x86_init_rdrand, which draws eight values, requires at least five changes between successive values, and on failure clears the feature bits for RDRAND and RDSEED. <https://github.com/torvalds/linux>
 - [44] Linux kernel source, arch/x86/kvm/vmx/vmx.c, arch/x86/kvm/emulate.c, arch/x86/kvm/svm/svm.c, arch/x86/lib/kaslr.c and arch/x86/include/asm/stackprotector.h. <https://github.com/torvalds/linux>
 - [45] Linux kernel source, arch/x86/include/asm/svm.h and arch/x86/include/uapi/asm/svm.h, whose intercept vector definitions and exit codes contain no entry for RDRAND or RDSEED. <https://github.com/torvalds/linux>
 - [46] M. Radev, M. Morbitzer, “Exploiting Interfaces of Secure Encrypted Virtual Machines”, *ROOTS’20: Reversing and Offensive-oriented Trends Symposium*, ACM, 2020, Section 4.2 (Early kernel entropy generation). <https://arxiv.org/abs/2010.07094>
 - [47] B. Petkov, “early x86 unseeded randomness”, Linux kernel mailing list, 14 August 2017, on initialising the boot CPU stack canary before the random driver is available. <https://lore.kernel.org/lkml/>
 - [48] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing”, *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, 2004.
 - [49] Intel, *Intel Trust Domain Extensions (TDX) Module Base Architecture Specification*, revision of April 2025. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html>
 - [50] Google Project Zero and Google Cloud Security, *Intel Trust Domain Extensions (TDX) Security Review*, April 2023. <https://google.github.io/security-research/pocs/intel/tdx/> and https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf
 - [51] Intel, *Data Center Attestation Primitives (DCAP)* and the TDX quote generation and verification documentation. <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html>
 - [52] AMD, *SEV Secure Nested Paging Firmware ABI Specification*, Publication No. 56860, covering attestation reports, VCEK and VLEK derivation. <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf>
 - [53] SUSE, *SUSE Linux Enterprise Server 15 SP7 Virtualization Guide*, chapter on AMD SEV-SNP. See also the OpenStack Nova specification on SEV-SNP libvirt support, which documents live migration and suspend as unsupported for protected guests.
 - [54] AMD, *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*, white paper, January 2020. <https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf>
 - [55] Microsoft, *About Azure Confidential VMs* and the accompanying confidential virtual machines FAQ, Microsoft Learn, documenting live migration for SEV-SNP protected guests. <https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview>
 - [56] J. A. Donenfeld, “x86/coco: Require seeding RNG with RDRAND on CoCo systems”, Linux kernel commit 99485c4c026f024e7cb82da84c7951dbe3deb584, commit message.
 - [57] Linux kernel documentation, *The Definitive SEV Guest API Documentation*, on SEV-SNP CPUID enforcement against a PSP-measured CPUID page, and *Add Secure TSC support for SNP guests*, on timestamp counter parameters negotiated with the AMD Secure Processor. <https://docs.kernel.org/virt/coco/sev-guest.html>
 - [58] AMD, *AMD Server Vulnerabilities*, Security Bulletin AMD-SB-3014, covering CVE-2024-21977.
 - [59] Linux kernel documentation, *The Kernel’s Command-Line Parameters*, admin-guide/kernel-parameters, entries for random.trust_cpu and random.trust_bootloader.
 - [60] AMD, *Secure VM Service Module for SEV-SNP Guests: Guest Communication Interface*, Publication No. 58019, Rev. 1.00, July 2023. <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf>
 - [61] COCONUT-SVSM project, *Development Plan Overview*, project documentation. <https://coconut-svsm.github.io/svsm/developer/DEVELOPMENT-PLAN/>
 - [62] NIST, *SP 800-22 Rev. 1a: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 2010. <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf>
 - [63] R. G. Brown, D. Eddelbuettel, D. Bauer, *Dieharder: A Random Number Test Suite*. <https://webhome.phy.duke.edu/~rgb/General/dieharder.php>
 - [64] NIST CMVP, *Entropy Certificates E97 and E98: Thales Luna HSM*. <https://csrc.nist.gov/projects/cryptographic-module-validation-program/entropy-validations/certificate/98>
 - [65] NIST, *SP800-90B EntropyAssessment*, commit 87c104d0ed4cbc96103e7b8b38d6f2c7e0a6b289, 26 May 2026, built from source as no release tag covers this revision. https://github.com/usnistgov/SP800-90B_EntropyAssessment
 - [66] S. Zhu, Y. Ma, T. Chen, J. Lin, J. Jing, “Analysis and Improvement of Entropy Estimators in NIST SP 800-90B for Non-IID Entropy Sources”, *IACR Transactions on Symmetric Cryptology*, 2017(3), pp. 151-168. <https://tosc.iacr.org/index.php/ToSC/article/view/769>
 - [67] NIST, *Comments Received on NIST SP 800-90B*, May 2016. <https://csrc.nist.gov/files/pubs/sp/800/90/b/final/docs/comments-received-sp800-90b-may2016.pdf>
 - [68] M. Johansson, *AIS 31 Test Suite from BSI, Version 1.0*, GitHub archive. <https://github.com/mjosaarinen/ais31-testsuite-v1.0>
 - [69] g-g-sakura, *AnotherAIS31TestTool: Conformance test tool of random numbers based on BSI AIS 20/31*, GitHub. <https://github.com/g-g-sakura/AnotherAIS31TestTool>