Nitro Isolation Engine Whitepaper

Published June 2026

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers, or licensors. AWS products or services are provided "as is" without warranties, representations, or conditions of any kind, whether express or implied.

© 2026, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Contents

Abstract

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides secure, flexible compute capacity in the cloud. The AWS Nitro System is the underlying foundation for all modern EC2 instances and has been designed from the start to provide zero operator access to customer data and strong isolation between customer instances. This whitepaper provides a detailed description of the Nitro Isolation Engine, a formally verified enhancement to the Nitro Hypervisor which raises the bar by providing mathematical rigor to the isolation of the Nitro System.

Introduction

Every day, customers around the world entrust Amazon Web Services (AWS) with their most sensitive applications. Keeping our customers' workloads secure and confidential while helping them meet their security, privacy, and data protection requirements is a top priority. We've invested in rigorous operational practices and security technologies that meet and exceed even our most demanding customers' data security needs.

The development of the AWS Nitro System has been a multi-year journey to reinvent the fundamental virtualization infrastructure of Amazon Elastic Compute Cloud (Amazon EC2). In 2017, we launched the Nitro System, the first cloud infrastructure designed to provide zero operator access to customer data. The Nitro Hypervisor, by design, has no networking stack, no general-purpose filesystem implementations, and no peripheral device driver support, and includes neither a shell nor any type of interactive access mode. The Nitro System is designed so that there is no mechanism for any system or person to sign in to EC2 Nitro hosts, read instance memory, or access customer data stored on local encrypted instance storage or remote encrypted Amazon Elastic Block Store (Amazon EBS) volumes. We have shared these innovations with the industry through whitepapers, and third parties such as NCC Group who validated our design in their 2023 independent architecture review. The Nitro controls that prevent operator access are so fundamental to the Nitro System that we've added them to our AWS Service Terms, which are applicable to anyone who uses AWS.

The Nitro Isolation Engine represents the next step in our continual commitment to raising the bar for our customers. One of the primary responsibilities of the AWS Nitro System is to isolate instances from each other and from AWS operators. The Nitro Isolation Engine is a purpose-built component within the Nitro System that compartmentalizes responsibility for enforcing this isolation and demonstrating it with

mathematical rigor. It's currently available on Graviton5 instance types and is responsible for enforcing isolation between virtual machines (VMs) including mediation of all access to VM memory, CPU register state, and I/O devices, through a minimal set of APIs. The Nitro Isolation Engine uses formal verification, a technique to mathematically demonstrate that the hardware or software behaves as intended, and not only in specific test cases. This intensive verification technique establishes Nitro System as the first formally verified cloud hypervisor, pioneering a new standard for cloud security supported by mathematical proof.

This whitepaper describes the design and architecture of the Nitro Isolation Engine and explains how formal verification has been applied to prove its correctness and security properties.

Background: The Nitro System and virtualization security

When you launch a virtualized EC2 instance, specialized software programs hardware to securely partition a single physical server into multiple isolated VMs. The underlying virtualization infrastructure of all modern EC2 instances is the AWS Nitro System.

The Nitro System comprises three components:

- **Purpose-built Nitro Cards** – Hardware devices designed by AWS that provide overall system control and I/O virtualization independent of the main system board with its CPUs and memory.
- **The Nitro Security Chip** – Enables a secure boot process for the overall system based on a hardware root of trust, the ability to offer bare metal instances, and defense-in-depth that offers protection from unauthorized modification of system firmware.
- **The Nitro Hypervisor** – A deliberately minimized and firmware-like hypervisor designed to provide strong resource isolation and performance that's nearly indistinguishable from a bare metal server.

The Security Design of the AWS Nitro System whitepaper provides a detailed view of the design of these components and how they function together as part of the overall Nitro System architecture. This whitepaper focuses specifically on the enhancements to the Nitro Hypervisor introduced with the Nitro Isolation Engine.

The Nitro Hypervisor is designed to uphold the properties of confidentiality and integrity for all guest VMs. It contains both the *mechanism* to manage guests—such as page tables, context switching, and I/O virtualization—and the business logic *policy* encoding how those guests should be managed—such as scheduling, memory allocation, and resource orchestration. This coupling means the Nitro Hypervisor code base is too large to mathematically prove its correctness or security properties through formal verification.

The Nitro Isolation Engine addresses this challenge by introducing a fundamental new principle into the Nitro System: the separation of mechanism from policy through compartmentalization.

The Nitro Isolation Engine: Architecture and design

Compartmentalization: Separating mechanism from policy

The Nitro Isolation Engine is a minimal separation kernel that compartmentalizes the critical functionality of running a VM into a minimal trusted computing base (TCB). It exposes only a constrained set of validated APIs to the rest of the Nitro Hypervisor.

This compartmentalization enforces a strict separation of concerns:

- **The Nitro Isolation Engine (mechanism)** – Executes at the most privileged hypervisor Exception Level EL2. It's the only system component of the Nitro Hypervisor capable of directly provisioning and managing guests, and the only component capable of directly interacting with guest VM state. It controls the CPU page tables, the I/O Memory Management Unit (IOMMU), and the saved guest CPU state.
- **The deprivileged Nitro Hypervisor (policy)** – Deprivileged to Exception Level EL1, the rest of the Nitro Hypervisor now conceptually sits side-by-side with the guest VMs. It retains ownership of scheduling decisions, resource allocation, device emulation, the interrupt controller, guest creation, and other operational logic. To manage guests, the Nitro Hypervisor uses a restricted hypercall API provided by the Nitro Isolation Engine.

Note. An Exception Level (EL) is the Arm term for processor execution privilege. The Arm architecture, used by AWS Graviton Processors, defines exception levels with increasing privilege from EL0 (typically used for user-level applications), EL1 (for operating systems), and EL2 (for hypervisors).

The following diagram captures the system architecture of a Nitro Isolation Engine enabled server:

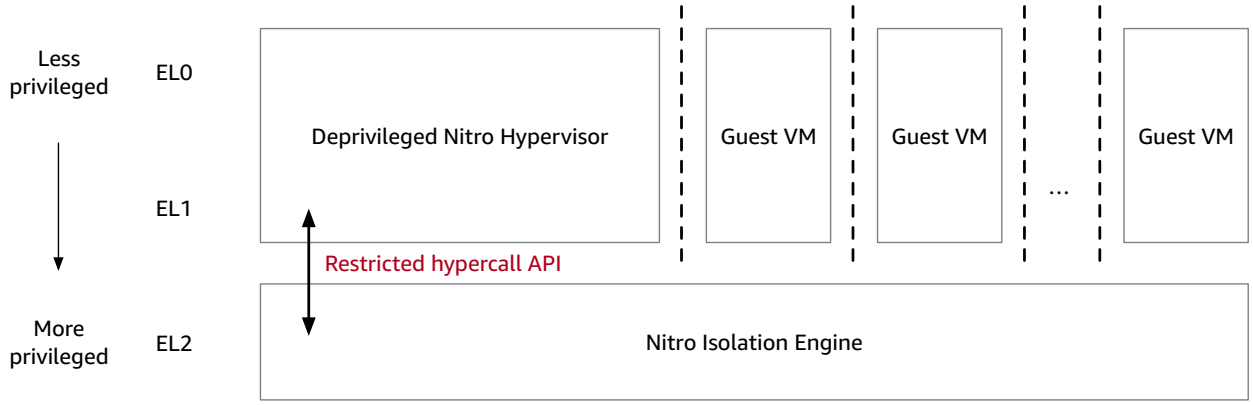

Figure 1: Architecture of a Nitro Isolation Engine enabled server

Under this architecture the Nitro Isolation Engine acts as a privileged intermediary between the rest of the hypervisor and all guests. The deprivileged component of the Nitro Hypervisor can't access guest VM state.

The hypercall API

The Nitro Isolation Engine exposes a narrow, well-defined hypercall API that the deprivileged component of the Nitro Hypervisor uses to manage VMs indirectly. Every hypercall API call is defined by a Machine Readable Specification (MRS) that captures input registers, output registers, error conditions, and the permissible state transition on success.

This API covers the full lifecycle of a VM:

- **VM creation and configuration** – Creating a new VM, allocating memory and metadata pages, and configuring the VM's initial state.
- **vCPU management** – Creating virtual CPUs, running virtual CPUs (entering the guest), and handling guest exits.

- **Memory management** – Mapping and unmapping memory pages for a guest, with strict enforcement that pages are scrubbed (zeroed) before reuse, so that no residual data from a previous guest can be observed.
- **I/O device management** – Configuring the IOMMU to only allow devices assigned to a guest VM to access that VM's memory.
- **Live update and live migration** – Supporting operational processes that update the Nitro Isolation Engine itself—or move a running guest from one physical server to another—while preserving security invariants.

The strict and minimal nature of this API is what makes formal verification feasible. Every interaction between the rest of the Nitro Hypervisor and guest state is mediated through this narrow interface, and the behavior of every API call is precisely specified and verified.

Memory isolation

The Nitro Isolation Engine controls the page tables that govern memory access. These page tables determine which physical memory pages each guest can access, and they're entirely under the control of the Nitro Isolation Engine—the rest of the Nitro Hypervisor can't modify them directly.

The engine enforces several critical invariants:

- **Single ownership** – At any given time, a page of physical memory is owned by exactly one principal in the system (a guest VM, the Nitro Isolation Engine, or the deprivileged Nitro Hypervisor).
- **Scrub-before-reuse** – When a memory page transitions from one owner to another, its contents are zeroed before the new owner can access it. This prevents information leakage between guests or from a guest to the hypervisor.
- **Controlled declassification** – A small, carefully audited set of register values and state elements are permitted to flow between a guest and the deprivileged component of the Nitro Hypervisor (for example, the information needed to handle a guest exit). These declassification policies are explicitly specified as part of the MRS and minimized.

I/O device isolation

EC2 instances use Nitro Cards for high-performance storage (EBS) and networking (ENA/EFA). These devices perform Direct Memory Access (DMA) to read and write memory on behalf of the instance. The Nitro Isolation Engine owns the System Memory Management Unit (SMMU), which is the IOMMU for the Arm architecture. The SMMU allows the Nitro Isolation Engine to enforce per-device access control at the page granularity.

Each guest instance is assigned exclusive access to specific devices from the Nitro Cards—no device is assigned to more than one guest—and those devices require access to the guest's memory to perform I/O operations on its behalf. The Nitro Isolation Engine programs the SMMU to grant each device access only to the memory belonging to its assigned guest, preventing any device from reaching memory outside that boundary.

Interrupt controller isolation

The Generic Interrupt Controller (GIC) is the interrupt controller for the Arm architecture. The deprivileged Nitro Hypervisor retains ownership of the GIC. As part of its normal operation, the GIC requires physical access to memory for configuration state (such as, interrupt status and priority). To provide isolation, we designed Graviton5 with the GIC placed behind an SMMU. Similar to I/O device isolation, the Nitro Isolation Engine programs this SMMU to grant the GIC access only to the memory belonging to the deprivileged Nitro Hypervisor, preventing the GIC from reaching memory outside that boundary.

Built from the start for verification

The Nitro Isolation Engine was designed from the first day of development with formal verification as a primary consideration. The durable value of involving formal verification from the beginning is in early specification, which influences technical direction, and ensuring that the path to proof is smooth. In our experience, these results are much more difficult to obtain when verification is retrofitted. The co-design of implementation and verification is one of the engine's most distinctive characteristics and central to the quality of the results.

Design decisions were continuously made in consultation with the formal verification team from the AWS Automated Reasoning Group. This includes:

- **Minimal code base** – Every feature added to the Nitro Isolation Engine represents more code that must be reasoned about. The team has been judicious in what functionality the engine owns versus what remains in the rest of the Nitro Hypervisor, keeping the code base as small as possible.
- **Verification-friendly coding patterns** – The implementation follows coding guidelines that make formal reasoning tractable. For example, hypercalls are written around a database-style transaction primitive that handles the taking and releasing of locks in a strict two-phase locking discipline, which enables serialization of concurrent transactions and simplifies reasoning about concurrency.
- **Machine Readable Specification** – The shared MRS serves as a single source of truth for both the engineering and verification teams. Both Rust implementation code and proof definitions are generated from these machine-readable documents, providing consistency between what's implemented and what's verified.

Written in Rust

The Nitro Isolation Engine is implemented in Rust, a programming language that's well suited for systems software development because of its focus on performance and reliability. By design, Rust aims to prevent common categories of software defects, including:

- **Memory safety violations** – Memory safety issues such as buffer overflows, NULL pointer dereferences, use-after-free, and double-free bugs, are prevented by Rust's ownership system, borrow checker and runtime checking.
- **Missing error handling** – Rust's `Result` and `Option` types, combined with exhaustive pattern matching, make it difficult to ignore error cases or corner conditions in function definitions.
- **Data race conditions** – Accessing shared data concurrently without proper synchronization is a compile-time error in Rust, prevented by its type system's enforcement of either exclusive mutable access or shared immutable access.

These errors, long known to be a leading contributing factor that leads to security vulnerabilities in systems software written in C and C++, are prevented at compile time

by Rust's novel type system and ownership-based memory management. Rust combines static guarantees provided by the compiler with pervasive runtime checks, making it particularly well-suited to implementing trustworthy systems software.

The choice of Rust for the Nitro Isolation Engine eliminates entire classes of bugs by construction—before any formal verification is even applied. This creates a layered defense: the language itself prevents the most common vulnerability classes, and formal verification then establishes deeper properties about the functional correctness and security of the implementation.

Formal verification of the Nitro Isolation Engine

Formal verification

Software testing is an effective and important approach for finding bugs. However, even the most comprehensive test suites can only check a specific set of inputs and system states and the space of possible inputs and system states for nontrivial systems is intractably large.

Formal verification is fundamentally different. Formal verification uses mathematical assurance to demonstrate that a formal (that is, mathematically precise) model of a system satisfies a formal specification. The strength of this approach is that proof results are mathematically rigorous evidence that the system—at the level of abstraction captured by the formal model—is correct with respect to its specification. Properties established with this approach hold across *all possible behaviors* of the formal model: every input, every sequence of events, and every system state, not only the cases that were tested.

To illustrate the difference to testing, consider an API hypercall that takes three u64 (unsigned 64-bit) values. The number of possible test cases is 2^{192} (that is, $2^{64} \times 2^{64} \times 2^{64}$): one for every possible assignment to each input. With a testing approach, even with a rate of compute of a billion (10^9) test cases per second per core on a billion CPU cores, exhaustive testing of this input space would take longer than the age of the universe (approximately 13.8 billion years). Formal verification can be used to establish properties that hold over all possible inputs in seconds. The approaches are complementary. Testing gives empirical evidence of the real system but can't establish universal properties. Formal verification offers a tractable way to reason about all

possible behaviors of a system but at the level of abstraction captured by the formalization.

The power of formal verification can be shown by its results in real-world applications. The seL4 microkernel, a landmark project in operating-system verification, has had zero functional correctness defects in its verified code in over 15 years of testing, deployment, and use; while defects have been found in every class of its unverified code in that same period. The CompCert verified C compiler showed similar results: a research study from the University of Utah devoted approximately six CPU-years to finding compiler bugs using automated testing and didn't find a single wrong-code error in CompCert's verified components, while finding bugs in every other compiler tested. As the seL4 project says: everything that can go wrong *will* eventually go wrong—unless you can prove it can't.

> **Note.** AWS generally—and the Nitro System specifically—has embraced formal verification as a key tool for development and validation. For example, AWS applies formal methods to verify the memory safety properties of early-stage boot code in the Nitro System. We similarly apply formal methods to prove that the network facing API of the control message parsing implementation in the Nitro Controller card is free from memory safety errors regardless of any configuration file and any network input. Based on the value we found in these earlier targeted applications of formal verification in the Nitro System, we invested ambitiously by taking on a project of massively broader scope and complexity with the Nitro Isolation Engine.

Evaluating formally verified systems

The most important questions to evaluate when examining a formally verified system are:

- **What is the scope of the formal verification?** Need to determine how much of the system has been formalized, and, by extension, what remains to be verified.
- **What properties have been proven?** This is important to understand to evaluate the strength of the results. Formally verified properties span a range of properties, from functional correctness to memory safety.

- **What assumptions underpin the proof results?** All formally verified systems are subject to assumptions that must be carefully reviewed to ensure they are reasonable because every assumption is a place where reality can diverge from the proof results. This characteristic is inherent in all applications of formal verification of real-world systems because proof results show that a formal model satisfies a formal specification, and these formal objects are *assumed* to correspond to a real-world informal artifact.

Scope of the Nitro Isolation Engine formal verification

The formal verification currently covers the first three parts of the VM lifecycle as given by the hypercall API: VM creation and configuration, vCPU management, and memory management. This represents the core production features of the system and a foundation from which more features can be covered.

Proven properties

The formal verification of the Nitro Isolation Engine establishes four key properties.

1. **Confidentiality and integrity –** The confidentiality (no unauthorized read access) and integrity (no unauthorized write access) of guest VM private data.
2. **Functional correctness –** The implementation behaves exactly as specified by the Nitro Isolation Engine specification with no additional behaviors.
3. **Absence of runtime errors –** The absence of runtime errors such as unwraps of `None` option values in Rust, arithmetic overflows or panics.
4. **Memory safety –** The absence of memory safety violations such as buffer overflows, NULL pointer dereferences, and out-of-bounds accesses. While Rust's type system can be used to prevent some of these issues at compile time, the formal proofs provide an additional, independent layer of mathematical assurance that applies to both safe and unsafe Rust code.

> **Note.** The Rust language defines unsafe Rust as a way to express idioms, such as raw pointer manipulation and inline assembly, that fall outside the static checks of the Rust compiler.

Assumptions that underpin the proof results

The assumptions of the Nitro Isolation Engine proofs can be broadly characterized as corresponding to:

- The extent to which the formal specification captures the system's informal real-world requirements.
- The breadth and depth of the formal model and its fidelity with respect to the real-world artifact (that is, the deployed binary running on real hardware that faithfully implements the Arm architecture).
- The trustworthiness of the proof infrastructure.

Of these, the most important to understand are assumptions in the second bullet because these characterize the breadth and depth of the proof results and make explicit the components that must be trusted, such as the Rust compiler and underlying hardware.

A full discussion of the assumptions and their mitigations for the Nitro Isolation Engine formal proof results is given in Soundness: Scope, assumptions, and risks. Understanding this relies on a more detailed understanding of the formal verification, which we cover in the next sections.

Understanding the proof stack

The formal verification of the Nitro Isolation Engine spans multiple layers, connecting high-level security properties to a formal model of the implementation through a chain of mathematical proofs. The following diagram illustrates this structure:

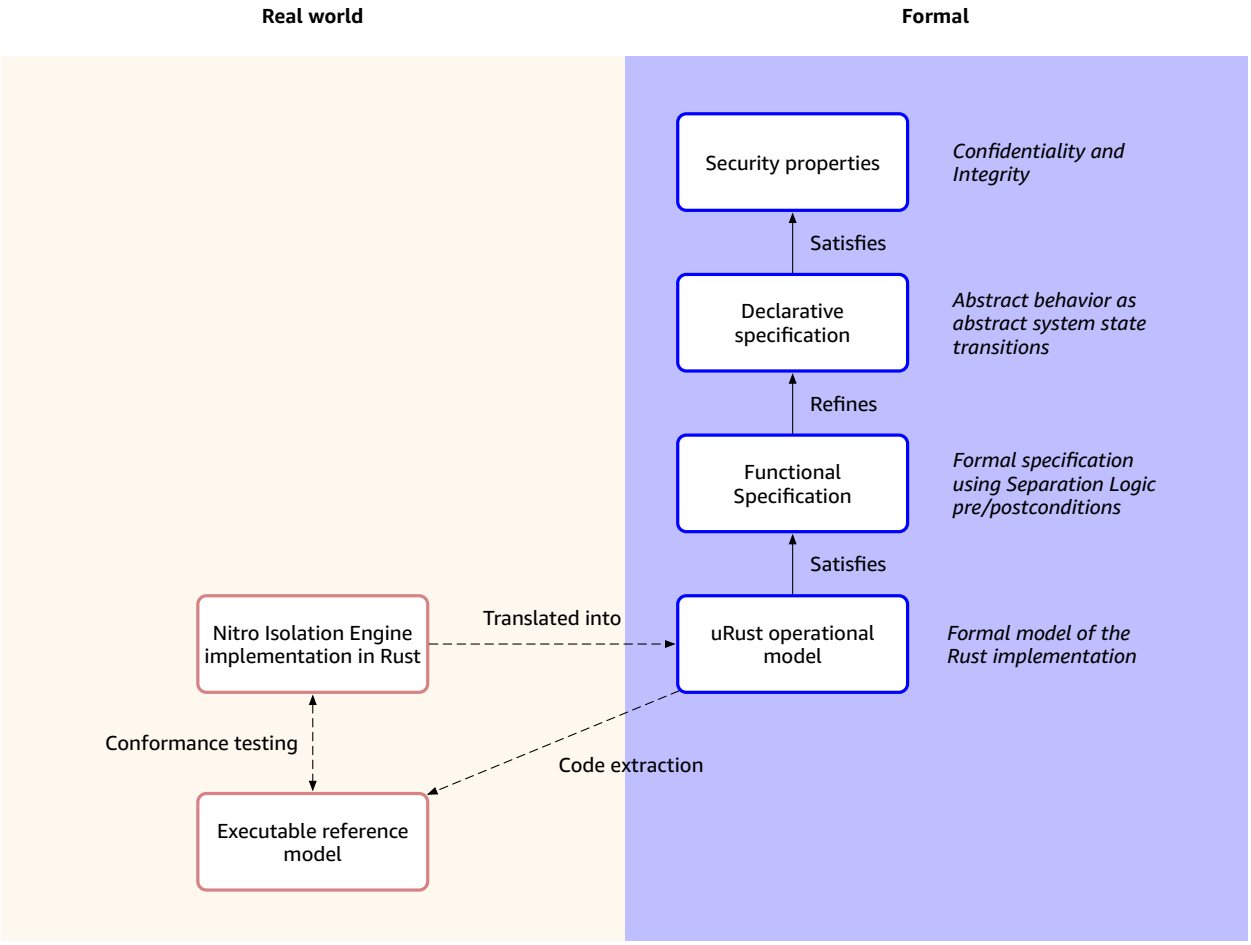

Figure 2: The formal verification of the Nitro Isolation Engine spans multiple layers

At the top of this stack sit the security properties of confidentiality and integrity. These properties are proven over an abstract declarative specification of the Nitro Isolation Engine that captures the behavior of system events as state transitions. A refinement proof then connects this abstract specification to a functional specification that is proven correct with respect to a formal operational model, written in a formalized fragment of the Rust language we call μRust (micro Rust). Finally, outside the proof chain, but vital for checking assumptions, the operational model is extracted as an executable reference model to connect the proof results empirically to the actual Rust binary through conformance testing, which is explained further in later sections.

This work is carried out in Isabelle/HOL, a mature and trustworthy interactive theorem proving system that has been developed continuously for nearly 40 years. Our formal verification results are, at the time of writing, approximately 330,000 lines of machine-checked mathematics.

The verification toolchain

μRust: A formal model of the implementation

We formalize the implementation of the Nitro Isolation Engine's in μRust, a restricted subset of the Rust programming language formalized in Higher Order Logic (the host logic of Isabelle/HOL) that's expressive enough to capture the engine's behavior while remaining amenable to formal reasoning.

The μRust representation of each hypercall closely follows the structure of the actual Rust implementation, making it straightforward to audit the translation between the two. The implementation of the Nitro Isolation Engine is deliberately restricted to a subset of Rust that μRust can express, avoiding features such as traits and dynamic dispatch that are unnecessary in our context and would complicate formal reasoning.

Specifications in separation logic

The expected behavior of a hypercall is captured as a *contract* with preconditions and postconditions—assertions about the state of the system before and after executing the hypercall.

Specifications are written in separation logic, a logic specifically designed for reasoning about programs that manipulate pointers and memory. Separation logic's *separating conjunction* allows us to express that different parts of the system state are disjoint (that is, non-aliasing and non-overlapping)—for example, that the memory used for the metadata of one guest is separate from the memory used for the metadata of another guest—in a way that composes cleanly across complex operations.

Weakest precondition calculus

To prove a μRust program correct with respect to its specification, we use a well-known idea from the verification literature known as weakest precondition calculus. Given a program and a desired postcondition, proof rules define the *weakest precondition*: the smallest set of conditions that guarantee executing the program will produce a state satisfying the postcondition. The proof obligation is then to show that the specified precondition implies this computed weakest precondition. We have developed extensive custom proof automation built on existing Isabelle automation to partially automate this process.

What's been proven in more detail

Functional correctness

Proofs of this property say that *if* the system state satisfies the system event's precondition *then* executing the *μ*Rust model of the system event will terminate in a state satisfying the postcondition, *and* the system invariant is preserved.

This first part of this result, which establishes functional correctness and termination, is also known as *total correctness*. The second part of this result establishes a system invariant: a property that is preserved by every system event that must hold before and after each system event (but temporarily might not hold during the execution of the system event). The system invariant captures important well-formedness properties of the internal system state, such as the relationship between a VM state and its vCPUs (for example, a VM can only have allocated vCPUs in certain VM states).

> **Note.** The additional proof obligation of termination, a problem known to be undecidable, makes total correctness strictly stronger than partial correctness, which only asserts "if the program terminates then the postcondition holds."

Total functional correctness is a strong property that implies the absence of whole classes of programming issues from memory safety and runtime errors, including:

- **No buffer overflows** – All indexes into arrays and slices are within bounds.
- **No NULL pointer dereferences** – All pointer dereferences are on valid references.
- **No arithmetic overflows** – Integer overflow and similar arithmetic errors that cause unexpected behavior can't occur.
- **No unhandled error conditions** – Every error path in the verified code is proven to be handled correctly, with no possibility of an unexpected panic or unrecoverable state.
- **No cyclic lock acquisition –** Locks are always acquired in a predefined global order. As a result, deadlock from cycles in lock acquisition can't occur.

- **Absence of non-termination –** All loops are bounded and execution terminates in a finite amount of time.

Refinement

The formal specifications are at two levels of abstraction: a high-level declarative specification where system events are specified as abstract system state transitions and a more detailed functional specification where system events are specified as separation logic pre- and post-conditions. The first is closer to the informal specification given by the MRS and the second is closer to the Rust implementation. A refinement proof connects the two specifications showing that if a system state refines an abstract system state M then executing the μRust model of the system event will terminate in a state that refines an abstract system state M' where $M \rightarrow M'$ is a state transition of the high-level declarative specification.

Confidentiality

Confidentiality means that a guest VM's private data can't be read by any unauthorized principal. We formalize this using a property called *noninterference*, expressed as an *indistinguishability preservation* property: for all abstract system states M and N and system events e if M and N are indistinguishable and M transitions to M' by e and N transitions to N' by e then M' and N' are indistinguishable.

The intuition is that the indistinguishability equivalence relation between two system states is a formal notion of the power of an observer. For example, an observer can always observe the contents of any physical pages mapped into its address space. Consider two abstract system states M and N that differ only in the contents of the private data of a particular guest whose confidentiality must be protected. An observer who isn't authorized to see that guest's data shouldn't be able to distinguish between these two states. Confidentiality holds if, after any system event (e, such as a hypercall), the two resulting abstract system states M' and N' remain indistinguishable to that observer. Intuitively, the observer has learned nothing new from the execution of the system event.

The Nitro Isolation Engine proofs specify an indistinguishability relation that combines the observational power of the rest of the Nitro Hypervisor and some number (possibly zero) of VMs. The definition captures architecturally visible state such as the contents of memory, general purpose registers and system registers, in addition to the internal state of the Isolation Engine, such as the number of allocated VMs. The definition includes

internal state because this information can be deduced by examining the sequence of system events.

An important consideration in the Nitro Isolation Engine confidentiality proofs is controlled declassification to allow for authorized information flows from guests to the rest of the Nitro Hypervisor for a predefined, static, and straightforward to reason about set of events. For example, on a guest data abort because of a write to an unmapped guest physical address, the deprivileged Nitro Hypervisor needs to learn the reason for the exit, the aborting guest physical address and the data to be written to properly handle the exit. The confidentiality proofs of a system event formalize the event's declassification policies to show that only these authorized information flows are permitted.

Integrity

Integrity means that a guest VM's private data can't be modified by any unauthorized principal. We formalize this as a *local respect* property: for all system events *e*, the private state of a principal doesn't observably change if that principal isn't the intended target of the system event *e*.

Conformance testing

The Nitro Isolation Engine proofs are established with respect to a formal model of the system whereas the real implementation is a physical artifact (that is, a binary running on hardware). Conformance testing connects these artifacts empirically. Traces taken from executing the real implementation are replayed on an executable reference model extracted from the formal model using Isabelle/HOL's code generation feature.

A key benefit of conformance testing is that it's *end-to-end*; meaning that execution traces are captured from running a compiled binary of the Nitro Isolation Engine on a realistic system. As a result, conformance testing also tests the Rust compilation process (that is, the compiler, assembler, and linker) in addition to validating proof assumptions about the semantics of μRust and hardware behavior.

Soundness: Scope, assumptions, and risks

No formal verification is absolute. Every verification effort links two formal objects—a specification and a model—each of which is assumed to correspond to something

informal and real. Transparency about these assumptions and the risks that arise when they fail is essential for customers evaluating the strength of our assurances.

This section describes what the Nitro Isolation Engine proofs establish, what they assume, and where the gaps lie. We see this as a living document: as our verification progresses, assumptions are reduced and gaps are narrowed.

Structure of the soundness argument

The relationship between the real system and our proofs can be understood through three gaps—places where assumptions bridge the formal and informal worlds and the trustworthiness of the proof infrastructure, as shown in the following diagram and described in the next sections:

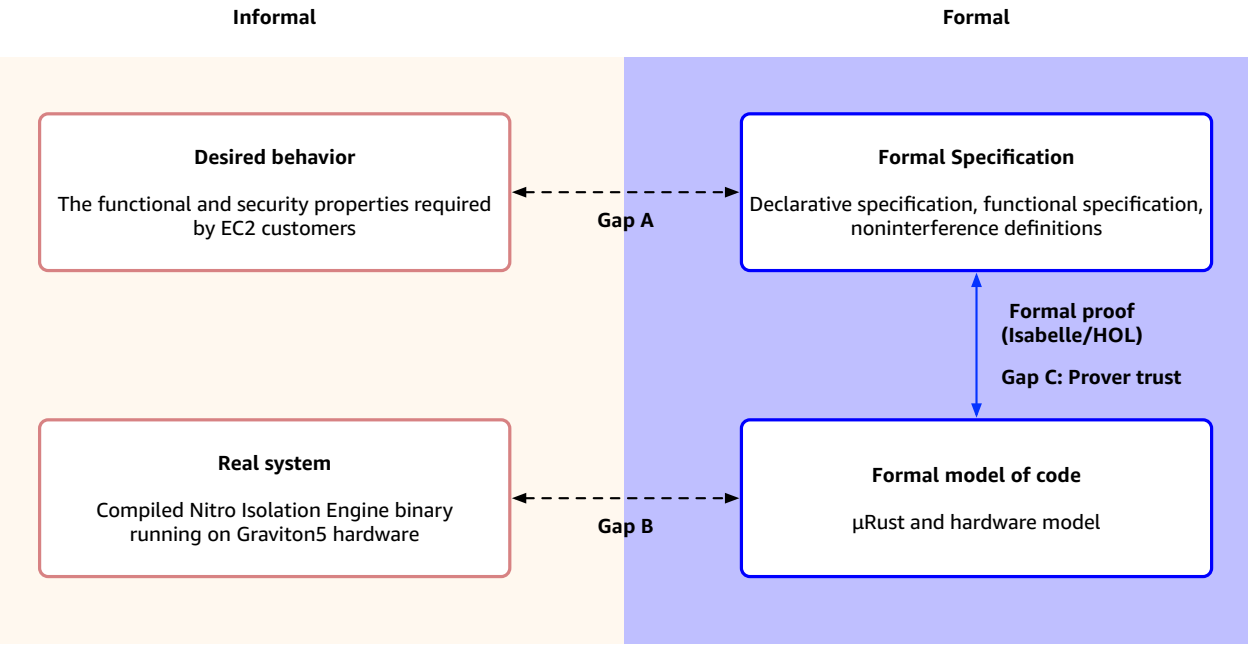

Figure 3: The relationship between the real system and our proofs

Gap A: Does the specification match the requirements?

Specification fidelity – The declarative formal specification defines the expected behavior of system events as abstract state transitions that match the informal specification from the MRS. Both the engineering team and the verification team use the same MRS documents, and common Rust implementation code and Isabelle/HOL definitions are generated from them. This co-generation approach reduces the risk of shared-mode specification failures. Additionally, the specifications are written in a

declarative, mathematical style that's not as complex as the implementation, making them amenable to human review.

Declassification policy – The security proofs must explicitly specify which information is permitted to flow between a guest and the deprivileged AWS Nitro Hypervisor (for example, the exit reason when a guest traps to the hypervisor). If this declassification policy is too permissive, the proofs would hold but the security properties would be weaker than intended. The declassification policy of a system event is deliberately minimized to only declassify information required for the proper functioning of the system. For auditability, each declassification policy is explicitly enumerated in the MRS and formally specified in the declarative specification.

Gap B: Does the formal model match the real system?

Proof scope – The formal model does not cover all parts of the real implementation. Errors in the implementation outside the boundary of the formal model could invalidate the proofs. The primary mitigation is engineering discipline and testing. Extending the proofs to cover all parts of the implementation is ongoing work.

***μ*Rust vs. compiled Rust binary** – The proofs reason about the *μ*Rust formalization of the source code, not the compiled machine code. The Rust compiler and LLVM backend are therefore in the trust boundary. Conformance testing is our primary mitigation for this risk. For example, a miscompilation resulting in different behavior between the compiled binary and the executable reference model would be caught.

Rust to *μ*Rust translation – The translation from Rust to *μ*Rust is manual and translation errors could change the meaning of the code. Conformance testing is our primary mitigation and ongoing work to automate the translation will further close the gap.

Model fidelity – The formal model includes abstractions of Rust semantics, the implementation and hardware. Errors in modeling these details could invalidate the proofs. For example, the formal model of the implementation's page table library captures the logical structure of multi-level page tables that is necessary to reason about the correctness of page table operations, but abstracts physical layout, such as page allocations and bit-precise page table entries. Conformance testing is our primary mitigation and extending the proofs to cover these aspects with greater fidelity is ongoing work.

Concurrency – The current proofs use sequential reasoning and don't model concurrent execution across multiple physical CPUs. The primary mitigation is that all internal shared data of the Nitro Isolation Engine is accessed using a transactional strict two-phase locking discipline. This provides serializability: every concurrent execution is equivalent to some sequential execution. To further mitigate this risk, the basic locking and transaction primitives used in this approach have been proven correct in two different concurrency-aware tools: Iris and Verus.

Micro-architectural state – The formal model is an architectural model: it captures the programmer-visible state of the machine (registers, memory, page tables) but doesn't model micro-architectural details such as caches, branch predictors, or speculative execution. The security proofs therefore establish properties about architecturally visible information flows. Side-channels that use micro-architectural-visible information flows aren't addressed by the proofs. These are mitigated through careful design and implementation, including secret-hiding, speculation barriers, and never sharing system resources such as L1 caches between different guest VMs.

More privileged software and hardware – The proofs assume correctness of systems code that is more privileged than the Nitro Isolation Engine, including bootloaders and firmware, and of the underlying hardware (such as Graviton5) with respect to its architectural specification.

Gap C: Is the proof infrastructure trustworthy?

Isabelle/HOL and the runtime – Isabelle/HOL has a small, auditable inference kernel. All theorems, no matter how complex the proof automation used to derive them, must ultimately be constructed through this kernel. Bugs in proof automation can't compromise soundness, they can only cause proofs to fail, not to succeed spuriously. This is a fundamental design property of the LCF (Logic for Computable Functions) architecture used by Isabelle/HOL.

Isabelle/HOL has been in continuous development and use for nearly 40 years, has been used for numerous high-assurance verification projects (including seL4), and has an excellent reputation for soundness. HOL as a logic has been studied extensively for consistency; we don't add any additional axioms to the base HOL logic in our proof, retaining this strong soundness guarantee. The runtime on which Isabelle executes is also part of the trust boundary; a compiler or runtime bug could in principle allow construction of a spurious theorem.

Separation logic – Our separation logic has been constructed and is sound by construction with respect to the semantics of our µRust fragment embedded in HOL.

Putting soundness in context

Formal verification is a rigorous bar. Transparency about soundness is necessary to clearly distinguish where the results provide mathematical assurance and where other mechanisms must be used. This transparency is a strength of formal verification and an important prerequisite for using proof within the larger discipline of building reliable trustworthy systems.

Secret-hiding and speculative execution side-channel defense

The formal proofs of the Nitro Isolation Engine establish that no unauthorized *architecturally visible* information flows can occur—that is, no direct read or write of memory or registers can leak or corrupt guest secrets. However, modern processors are also subject to *micro-architectural* side-channel attacks, such as Spectre and Meltdown, which exploit speculative execution to access state that would ordinarily be disallowed, observing the results through timing differences or cache behavior.

Rather than attempting to mitigate each individual side-channel attack as it's discovered, the Nitro Isolation Engine maintains the same general defensive *secret-hiding* technique employed by the Nitro Hypervisor.

What is secret-hiding?

The core principle of secret-hiding is simple: if privileged software doesn't have access to secrets, it can't leak them regardless of what micro-architectural side-channel might exist.

Secret-hiding relies on address translation to ensure that guest secret information (memory contents, register state) isn't mapped into the Nitro Isolation Engine's address space except during the brief intervals when the engine is executing on behalf of that specific guest.

During steady-state execution, guest VM memory isn't mapped into the address space of the Nitro Isolation Engine at all. The engine operates on its own private memory,

which contains no guest secrets. When a guest exits (for example, because of a hypercall or an interrupt), the engine saves the guest's register state into dedicated *secret pages* that are only mapped when executing in that guest's context, and then unmaps them before returning control to the Nitro Hypervisor.

This design helps ensure that at no point does the Nitro Isolation Engine hold a mapping to a guest's secrets while executing on behalf of another guest or the deprivileged Nitro Hypervisor. The window during which secrets are mapped is minimized to the brief period of the context switch itself.

Memory scrubbing

When guest memory pages are reclaimed (through the `VM.Unmap` hypercall), their contents must be scrubbed (zeroed) before they can be reused by another principal. This scrubbing is performed through temporary mappings in the secret page tables, so that the page contents are accessible only during the scrubbing operation and are immediately unmapped afterward. The formal proofs establish that scrub-before-reuse is always performed and that no page can be observed by a new owner before scrubbing is complete.

Attestation of host software

Formal verification proves that the Nitro Isolation Engine source code satisfies its specification. The secure and measured boot process of the AWS Nitro System enables AWS to remotely validate that the Nitro Isolation Engine code running on a particular Nitro System server is precisely the same verified code.

Attested boot

Graviton CPUs contains cryptographic secrets burned into the silicon during manufacturing. These hardware-rooted secrets form the foundation of an attested boot process that establishes a chain of trust from the physical device to the running software. Before any guest VMs are permitted to run on a server, the attested boot flow cryptographically confirms three conditions:

1. Authentic production hardware – The attestation proves that this is a legitimate production Graviton device, not a test, development, or counterfeit system.

2. Known firmware – The attestation confirms that the system has booted known, expected firmware at every stage of the boot sequence.
3. Expected Nitro Isolation Engine version – The attestation confirms that the Nitro Isolation Engine loaded on the system matches an expected, verified build before any guest workloads are allowed to execute.

The result of this process is an attestation report: a chain of certificates that roots back to the hardware secrets in the Graviton CPU. This chain provides cryptographic evidence that the entire software stack, from firmware through the Nitro Isolation Engine, is in the expected state.

Live update attestation

The Nitro Isolation Engine supports live update—the ability to replace the running engine with a new version without disrupting guest VMs. Attestation extends seamlessly across live updates:

1. When a live update occurs, a new certificate is created that identifies the updated version of the Nitro Isolation Engine.
2. This new certificate is signed with the key belonging to the previous certificate in the chain.
3. The previous key is then destroyed.

This mechanism does more than confirm the current measurements of the running software. It records the complete history of live updates the system has undergone, producing an append-only chain of cryptographic evidence. Each link in the chain is signed by its predecessor and the predecessor's key is irrevocably destroyed, designed to make it infeasible to forge or omit entries.

Change management

All changes to the Nitro Isolation Engine follow the same rigorous change management process as the broader Nitro System. Code changes are subject to multi-party review and approval, including review by an engineer with substantial Amazon EC2 tenure serving as a member of the Change Management Bar Raiser program. In addition to expert human review, all check-ins must pass automated quality and security checks that run under the control of a central build service. After reviews and automated checks are complete, signed binaries are deployed to the fleet through the standard EC2

deployment pipeline, rolling out in waves across Availability Zones and AWS Regions with continuous monitoring and automatic rollback on anomalous behavior.

Conclusion

The Nitro Isolation Engine represents a significant advancement in cloud security since the introduction of the Nitro System itself. By compartmentalizing the critical mechanisms that protect guest VM state into a minimal, Rust-based separation kernel, and subjecting that kernel to rigorous formal verification, we reinforce our consistent commitment to ensure that the Nitro System enables customers to build with the confidence that their data is isolated and inaccessible. Our commitment to raising the bar is ongoing, including expanding the proof scope, formally expressing secret-hiding, and incorporating more details of the underlying Arm architecture into the proofs.

We believe this level of transparency sets a new standard for how cloud providers can demonstrate openness, code quality, and formal verification.

Contributors

Contributors to this document include:

- J.D. Bean, Principal Security Architect, Amazon EC2
- Nathan Chong, Principal Applied Scientist, Automated Reasoning Group
- Dominic Mulligan, Principal Applied Scientist, Automated Reasoning Group
- Ali Saidi, VP/Distinguished Engineer, Amazon EC2

Document revisions

Date	Description
2026	First publication