Waseda University

School of Fundamental Science and Engineering

Demystifying Coconut-SVSM: An Architectural and Empirical Characterization of an AMD SEV-SNP Paravisor

Bachelor's Thesis

Author: Yoshiaki Sato
Student ID: 1W22CF08
Supervisor: Professor Keiji Kimura
Submission: July 2026

Abstract

A Trusted Platform Module (TPM) provides security functions used for measured boot, protected keys, sealed secrets, platform identity, and attestation. Virtual machines need equivalent functionality, so cloud systems commonly provide each VM with a virtual TPM (vTPM). In conventional virtualization, the vTPM can run in the trusted host stack. A confidential virtual machine changes this assumption by protecting guest memory and processor state from the host itself. Keeping security-sensitive vTPM state in that host would weaken the confidential VM's trust model.

AMD Secure Encrypted Virtualization with Secure Nested Paging provides Virtual Machine Privilege Levels that allow a privileged service layer to run inside the confidential VM while remaining isolated from the ordinary guest operating system. Coconut-SVSM is a prominent open-source implementation of this Secure VM Service Module model and includes a vTPM at VMPL0. This thesis presents an implementation-centered architectural and empirical characterization of that stack on a physical AMD EPYC 9175F server. It combines source-level tracing with measurements of cold-boot behavior, guest-observed SVSM protocol-call latency, and Linux-visible memory accounting, then evaluates the vTPM through the standard Linux TPM interface.

The Linux call-path analysis shows how an unmodified userspace TPM command is translated by the `tpm_svsm` driver into an SVSM request, issued through the SVSM Calling Area and a VMGEXIT, and returned to the ordinary TPM subsystem. Across three representative commands, the median duration of an observed `svsm_perform_ghcb_protocol()` call ranged from 35.9 µs to 49.1 µs. Aggregate traced SVSM time per command ranged from approximately 112 µs to 147 µs, below approximately 0.5% of the observed 30 ms userspace command time. Linux reported only 28 KiB less total memory and about 252 KiB more reserved memory in the SVSM configuration, both below 0.004% of an 8 GiB guest. The vTPM completed a representative PCR-bound secret-release workflow and rejected the same policy after the PCR state changed.

Repeated cold-boot measurements showed median ready times of 9.386 s, 14.516 s, and 20.504 s for the normal, SNP, and SNP with SVSM configurations, respectively. The SVSM configuration added 5.988 s over SNP. The dominant interval occurred before the Linux banner and contained approximately 2,100 lines of measured-boot and TPM debug output, so the result characterizes the complete debug stack rather than an intrinsic VMPL or vTPM penalty.

Together, the results show that the evaluated Coconut-SVSM stack moves useful vTPM functionality into the CVM without making steady-state protocol latency or guest-visible memory capacity the main practical concern. Its principal limitations instead appeared in debug-instrumented startup, early-boot TPM availability, persistent state, and the integration of TPM evidence with remote attestation.

Acknowledgments

I would like to thank Professor Keiji Kimura and Professor Hironori Kasahara for his supervision and guidance throughout this research. I am also grateful to the developers of Coconut-SVSM and the broader Linux confidential-computing community for making the source code, specifications, and development discussions openly available. Their work made it possible to study a rapidly evolving systems stack at the level of concrete implementation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why confidential virtual machines need a trusted vTPM

A Trusted Platform Module (TPM) is a security component that protects cryptographic keys and provides a hardware-backed record of platform state. Its Platform Configuration Registers (PCRs) accumulate measurements of firmware, bootloaders, kernels, and other components. Software can seal a secret to an expected PCR state so that the TPM releases it only when the required measurements are present. TPMs also support random generation, protected objects, platform identity, and attestation [1]. These functions are used in systems such as measured boot, disk encryption, credential protection, integrity verification, and trusted provisioning.

Virtual machines need many of the same functions. A physical TPM, however, belongs to the host platform and cannot normally provide an independent security identity and private state store to every VM. Cloud platforms therefore emulate a separate virtual TPM (vTPM) for each guest. Existing operating systems and applications can access it through the same TPM command model used for a physical device. In conventional virtualization, placing this vTPM in QEMU or another host component is a natural design because the host is already trusted.

Confidential virtual machines change that relationship. A customer still controls the software inside a VM, while the provider controls the hypervisor, host operating system, firmware, and physical infrastructure. Confidential computing uses hardware-enforced trusted execution environments to protect data while it is in use, reducing the need to trust much of this provider-controlled software. AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP), for example, encrypts and isolates guest memory and processor state and protects guest memory mappings [2, 3].

This creates a direct conflict for a conventional host vTPM. The component responsible for protecting keys, recording measurements, and authorizing secret release would remain accessible to the same host that the CVM is designed not to trust. A malicious or compromised provider could potentially inspect vTPM secrets, modify its state, or interfere with the evidence presented to the guest. Moving the vTPM into the ordinary guest kernel would remove it from the host, but would then expose it to a compromised guest operating system.

AMD's Virtual Machine Privilege Level mechanism provides a third location. SEV-SNP defines four privilege levels, VMPL0 through VMPL3, inside one protected guest address space. VMPL0 is the most privileged, and hardware-enforced memory permissions can prevent a less privileged guest kernel from accessing VMPL0-private pages. The Secure VM Service Module (SVSM) specification uses this mechanism to place a small service environment inside the CVM trust boundary while keeping it separate from the ordinary guest [4]. The host still schedules transitions, but the vTPM implementation and its private state can remain encrypted and protected within the CVM.

Coconut-SVSM is an open-source implementation of this model. It was initiated by SUSE, is hosted through the Linux Foundation's Confidential Computing Consortium, and receives contri-

butions from organizations active in confidential computing [5,6]. Its Rust-based environment runs at VMPL0 and provides services including a vTPM to a lower-privilege Linux guest. The complete stack spans KVM, QEMU, EDK2/OVMF, an IGVM launch image, Coconut-SVSM, and the guest kernel. This makes the vTPM a concrete example through which to study both the security value and the engineering cost of placing trusted services inside a CVM.

1.2 Problem statement

Placing the vTPM inside the CVM resolves an important trust problem, but it also introduces practical questions. A privileged layer adds software to the trusted computing base. Requests must cross a new protection boundary. Firmware and measured-boot paths become more complex. Memory must be reserved for the service layer and its communication structures. Guest kernels require drivers that translate established TPM interfaces into the SVSM protocol.

These costs are difficult to infer from the architecture alone. A VMPL transition might be inexpensive compared with a complete TPM command, or it might dominate a fine-grained service. A paravisor might occupy a large region of guest memory, or its guest-visible effect might be negligible. A vTPM might enumerate successfully while failing common policy operations. A boot might be slowed primarily by the SVSM itself, by confidential-guest initialization, by firmware integration, or by measurement instrumentation.

The available specifications define the required protocols, while prior research proposes secure vTPMs and other VMPL-based services. They leave an implementation-level gap between the architectural design and the behavior of a current open-source stack assembled on physical hardware. In particular, they do not show whether the Coconut-SVSM vTPM works through ordinary Linux tools, where its observable costs appear, or how a command crosses from the guest into VMPL0.

This thesis addresses that gap by studying Coconut-SVSM as a complete systems artifact. It connects the trust-model benefit of an in-CVM vTPM to its Linux integration, representative functionality, call-path latency, memory accounting, and cold-start behavior. The result is an empirical account of what the evaluated implementation provides today and which engineering concerns are most important for its continued development.

1.3 Research questions

The study is organized around three research questions.

RQ1: What guest-visible operational costs are observed when Coconut-SVSM is added to an AMD SEV-SNP CVM? The evaluation considers cold-boot behavior, latency in the guest-to-SVSM protocol path, and Linux-visible memory accounting. These metrics cover a deployment-level cost, a fine-grained service-boundary cost, and a static capacity effect.

RQ2: To what extent does the tested SVSM-backed vTPM support representative TPM 2.0 workflows through standard Linux tools? The functional study includes device discovery, capability queries, random generation, PCR reads and extensions, object creation, sealing and unsealing, and both successful and unsuccessful PCR-policy cases.

RQ3: How does a TPM command travel from Linux userspace to the VMPL0 service boundary? Source-level tracing follows the command from `tpm2-tools` through the Linux TPM subsystem and `tpm_svsm` driver, into the SVSM call protocol, and down to the guest-side VMGEXIT instruction sequence.

1.4 Scope and thesis statement

This work is an implementation-centered architectural and empirical characterization of one Coconut-SVSM configuration. The evaluated system uses an AMD EPYC 9175F host, an 8 GiB SEV-SNP guest, Coconut-SVSM commit `44f639d2`, and a Linux 7.0 guest with the SVSM vTPM driver enabled.

The central thesis is:

Thesis statement

In the evaluated stack, Coconut-SVSM exposed a Linux-compatible vTPM that supported representative PCR and sealing workflows after driver registration. The guest-observed SVSM protocol path took tens of microseconds per call and was a small component of the tested userspace command time, while changes in Linux-visible memory accounting were negligible for an 8 GiB guest. Repeated boot measurements showed a 5.988 s median complete-stack difference over SNP, concentrated in a verbose pre-kernel measured-boot path.

The findings describe the evaluated configuration and the representative workflows used in the experiments. The functional evaluation focuses on PCR and sealing behavior, while the latency measurement covers the complete guest-observed SVSM protocol region, including more than the hardware-level VMPL transition.

1.5 Contributions

The original contribution of this thesis is a combined architectural and empirical characterization of a recent Coconut-SVSM implementation on real SEV-SNP hardware. Rather than proposing a new vTPM design, it makes the behavior of the existing open-source stack concrete through a reproducible deployment, source-level tracing, measurements, and an end-to-end policy test. This thesis makes six specific contributions.

1. It documents a reproducible Coconut-SVSM environment across host Linux, QEMU, IGVM, EDK2/OVMF, guest Linux, and the SVSM itself, including exact repository commits and launch configurations.
2. It measures 30 cold starts across normal, SNP, and SNP with SVSM configurations, reports robust distribution statistics, and localizes the main SVSM-associated difference to a high-volume pre-kernel debug interval.
3. It provides a source-guided explanation of the Linux guest-to-SVSM vTPM path. This connects the familiar `/dev/tpmrm0` interface to the Calling Area, GHCB or MSR transport selection, and `rep; vmmcall` boundary.
4. It measures the guest-observed duration of `svsm_perform_ghcb_protocol()` for three representative TPM commands over 50 command executions each. It reports both per-call latency and aggregate traced time per userspace command.
5. It characterizes the change in Linux-visible memory accounting between an SNP guest and an SNP with SVSM guest, while distinguishing that metric from the complete resident memory of the paravisor.
6. It evaluates the vTPM using a semantically meaningful security workflow. A secret is released when the current PCR value satisfies the policy and is blocked after the PCR is changed. This negative case demonstrates enforcement, not merely successful device enumeration.

1.6 Main findings

The main findings are summarized in Table 1.1.

Table 1.1: Summary of the empirical findings and their correct interpretation.

Dimension	Observation	Supported interpretation
Boot behavior	Median ready times across ten runs: normal 9.386 s; SNP 14.516 s; SVSM 20.504 s.	The SVSM configuration added 5.988 s over SNP. Most of the difference coincided with high-volume measured-boot debug output before Linux entry.
Protocol latency	Median call latency of 35.9 µs to 49.1 µs; aggregate 112 µs to 147 µs per command.	The traced guest-to-SVSM region is small relative to the approximately 30 ms command time for the three tested commands.
Memory accounting	28 KiB lower reported total memory and about 252 KiB more reserved memory.	The guest-visible capacity effect is below 0.004%; total VMPL0-private allocation remains a separate measurement question.
vTPM functionality	Standard commands, sealing, matching PCR policy, and changed-PCR rejection behaved as expected.	The tested stack supports a representative local PCR-bound secret-release workflow; conformance and broader feature coverage require separate testing.

1.7 Thesis organization

Chapter 2 introduces confidential virtual machines, SEV-SNP, VMPLs, SVSM, and TPM concepts. Chapter 3 explains the Coconut-SVSM stack and traces the Linux vTPM command path. Chapter 4 defines the environment, comparisons, metrics, and validity controls. Chapter 5 evaluates boot behavior, protocol-call latency, and guest-visible memory. Chapter 6 presents the vTPM functional evaluation. Chapter 7 synthesizes the security value, practical costs, limitations, and future work. Chapter 8 relates the study to prior CVM measurement, VMPL service frameworks, verified security modules, confidential vTPM research, and the likely requirements of future Coconut-SVSM use cases. Chapter 9 concludes the thesis.

Chapter 2

Technical Background

2.1 Trusted execution environments and confidential virtual machines

A trusted execution environment protects code and data from software outside a defined trust boundary. Earlier widely deployed TEEs, such as application enclaves, required software to be partitioned so that a small sensitive component could run inside a restricted environment. Confidential virtual machines instead aim to protect a complete VM. This preserves familiar operating systems, libraries, and deployment models while changing which infrastructure components must be trusted.

The central threat is a privileged but untrusted host. In an ordinary VM, the hypervisor controls second-level page tables, observes guest-physical mappings, handles exits, and can access host memory that contains guest state. A compromised hypervisor is therefore normally able to read or alter the VM. A CVM uses processor and firmware mechanisms to encrypt memory with a guest-specific key, protect saved register state, and validate address translations. The host still allocates resources and schedules virtual CPUs, but it should not learn private guest contents merely by exercising that control.

This distinction is important: confidential computing primarily reduces trust in the provider-controlled software stack. It does not automatically make the guest operating system trustworthy, eliminate side channels, guarantee availability, or validate every application. The host may still delay, pause, or terminate a CVM. The guest itself may contain exploitable code. Attestation can identify a launched configuration, but it does not prove that the configuration is free of vulnerabilities.

2.2 Evolution of AMD SEV

AMD SEV evolved in stages [2]. The original SEV feature encrypted each VM's memory with a distinct key. SEV-ES extended the protection to processor register state by encrypting the Virtual Machine Save Area when a virtual CPU exits. SEV-SNP added integrity protection for guest memory mappings and a stronger model for page ownership and state [3, 7].

2.2.1 Memory encryption and protected register state

Memory encryption prevents the host from directly interpreting private guest pages. The encryption key is associated with the address-space identifier assigned to the guest and is managed below the hypervisor's normal control. SEV-ES complements this by protecting the register image stored on an exit. Without protected register state, a hypervisor could observe sensitive data in general-purpose registers even when memory is encrypted.

Encryption by itself is not sufficient. A malicious hypervisor controls nested page tables and

could attempt to remap ciphertext, replay an older page, or alias one guest-physical address to an unexpected system-physical page. SEV-SNP addresses this class of problem using the Reverse Map Table.

2.2.2 Reverse Map Table

The Reverse Map Table, or RMP, records security metadata for protected system-physical pages. Its entries include ownership and mapping information enforced by hardware. The RMP therefore constrains how the hypervisor can assign or remap private guest pages. Page-state transitions require a coordinated protocol between guest, hypervisor, and secure processor.

The guest also validates private pages using the `PVALIDATE` instruction. In an ordinary SNP guest, the guest kernel can perform many of these operations directly. When the guest runs below VMPL0, some operations become architecturally restricted. An SVSM can proxy them through its core protocol.

2.2.3 Attestation

SEV-SNP supports signed attestation reports that bind report data to a measured guest launch and platform security version. The firmware ABI defines the report structure, guest messaging, and endorsement-key model [7]. A remote verifier can use the certificate chain and report signature to check that evidence originated from an AMD platform with a stated trusted computing base.

Attestation is related to, but distinct from, TPM-based measured boot. The SNP report describes the hardware-protected guest launch and platform state. A TPM's Platform Configuration Registers can accumulate later measurements of firmware, bootloaders, the kernel, configuration, and runtime components. A complete system may need to bind these two roots of evidence.

2.3 Virtual Machine Privilege Levels

2.3.1 A privilege dimension inside the CVM

SEV-SNP defines four Virtual Machine Privilege Levels. VMPL0 is the most privileged and VMPL3 is the least privileged. The numbering is the reverse of the intuitive meaning of a larger number, so it is clearer to say "more privileged" or "less privileged" rather than "higher" without qualification.

VMPLs are separate from x86 Current Privilege Levels, commonly called rings. A kernel may execute at CPL0 while its virtual CPU is assigned to VMPL2. Ring 0 then gives the kernel ordinary operating-system privilege within its layer, but it does not override the RMP permissions imposed by VMPL0.

RMP entries can define read, write, and execute permissions for numerically higher VMPLs. This makes it possible for VMPL0 to hold pages that the guest kernel cannot access. The SVSM specification expects the module and its initial data to occupy a contiguous guest-physical range that is accessible to VMPL0 but protected from less privileged VMPLs [4].

2.3.2 Why VMPLs are useful

The VMPL mechanism addresses a gap between two security models. Placing a service in the guest kernel protects it from the host but not from a compromised guest. Placing it in the host isolates it from the guest but trusts the provider. Placing it at VMPL0 can protect it from both the host software and a less privileged guest kernel, subject to the security of the processor, firmware, SVSM, and protocol.

Potential services include virtual TPMs, page validation, attestation mediators, secure variable stores, migration helpers, interrupt controllers, and protected monitors. The AMD

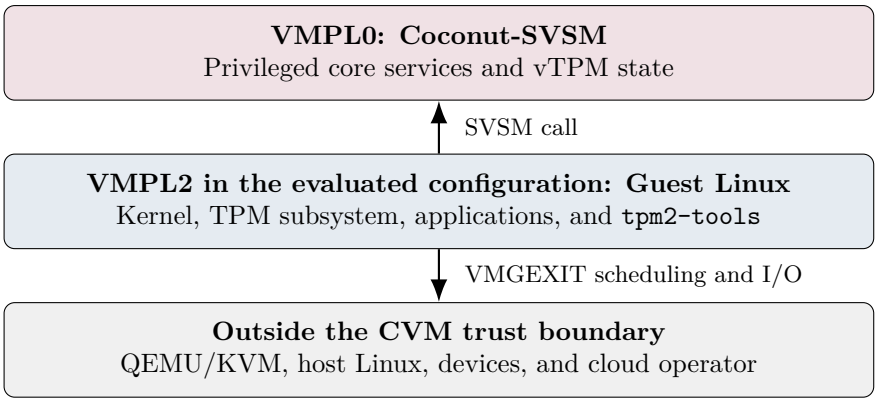

Figure 2.1: Simplified privilege structure of the evaluated SNP with SVSM guest. VMPL0 and the guest operating system share one confidential guest context but have hardware-enforced privilege separation.

SVSM specification currently defines core, attestation, vTPM, APIC-emulation, and UEFI-management-mode protocols [4]. An implementation may support only a subset.

2.4 Secure VM Service Module

2.4.1 Purpose and discovery

An SVSM is an execution environment for privileged modules inside an SNP guest. The specification deliberately focuses on guest communication rather than prescribing one internal design [4]. This permits several implementations to expose compatible services.

At boot, the SVSM publishes discovery data through fields in the SNP secrets page. These fields identify the base and size of the SVSM region, the address of the initial SVSM Calling Area, the maximum core-protocol version, and the VMPL at which the ordinary guest executes. A nonzero SVSM size indicates that an SVSM is present. The guest must reserve the SVSM range rather than treating it as normal RAM.

2.4.2 Calling Area and register convention

Each request uses both registers and an SVSM Calling Area. The Calling Area contains a `call_pending` byte and space for protocol data. The guest sets `call_pending` to one, prepares its registers, and executes VMGEXIT. RAX identifies the protocol and call, while RCX, RDX, R8, and R9 carry arguments in a convention similar to the Microsoft x64 ABI.

The pending byte helps the guest and SVSM reason about the untrusted host and is not merely a completion flag. The SVSM clears the byte when it accepts and completes a request. On return, the guest atomically exchanges the field and examines the old value. If it is still nonzero, the guest knows that the requested call did not complete. This cannot force the host to provide service, but it prevents the guest from silently accepting an unperformed request as successful [4].

2.4.3 GHCB and MSR transports

The VMGEXIT instruction transfers control out of the current guest context. The host must know what operation the guest requests. SEV-ES and SEV-SNP standardize this communication through the Guest-Hypervisor Communication Block [8].

Linux can issue an SVSM request using two related forms. When a GHCB page is available, it writes an exit code such as `SVM_VMGEXIT_SNP_RUN_VMPL` into that page and exposes its physical address through the GHCB MSR. During very early boot, or when a GHCB page is temporarily

unavailable, Linux can encode the request directly in the MSR protocol. Both paths still culminate in VMGEXIT. They are transport choices around the same logical request, not two independent vTPM designs.

2.5 Coconut-SVSM

Coconut-SVSM implements the SVSM concept as open-source software written primarily in Rust [5]. It contains a small kernel, architecture-specific startup code, service code, and build tooling. The project describes three long-term operating modes: an enlightened guest mode, a paravisor mode for guests with limited CVM awareness, and a service-VM mode [9]. The evaluated stack uses an enlightened Linux guest that is aware of SVSM and calls it directly.

The launch artifact uses the Independent Guest Virtual Machine format. IGVM packages the information required to construct a measured initial guest, including the Coconut-SVSM and OVMF images in this configuration [10, 11]. QEMU interprets the IGVM file and cooperates with KVM and the AMD Secure Processor to install the initial pages.

This packaging has an important evaluation consequence. The SNP baseline passes OVMF through QEMU's `-bios` option, while the SVSM configuration embeds OVMF in the IGVM artifact. A direct boot comparison therefore measures the deployment-level difference between complete configurations. It does not isolate only the execution time of the SVSM binary.

2.6 Trusted Platform Modules

2.6.1 State, measurement, and policy

A TPM is a stateful cryptographic component with standardized commands and objects [1]. It can generate random data, create keys, store or protect secrets, maintain monotonic state, and support attestation. Platform Configuration Registers (PCRs) are particularly important for boot integrity. PCR values are updated using the following operation

$$\mathrm{PCR}_{new} = H(\mathrm{PCR}_{old} \parallel d), \tag{2.1}$$

where d is the digest of a new measurement and H is the selected hash algorithm. The chained construction makes the final value depend on the ordered measurement history.

A sealed object can contain data whose release requires an authorization policy. A PCR policy asserts that selected PCRs must match expected values. Software first creates a policy digest from an expected PCR state, associates that policy with a sealed object, and later starts a policy session. If the current PCR state satisfies the policy, the TPM authorizes unsealing. If a measured component changes, the PCR value changes and the previous policy no longer succeeds.

2.6.2 Why a vTPM is difficult for a CVM

A virtual machine cannot normally access a unique physical TPM directly. Cloud platforms therefore emulate a TPM per VM. Traditional vTPMs are often host processes or devices managed by the virtualization stack. That design conflicts with a CVM threat model because the host can observe the vTPM's secrets and state.

An SVSM-backed vTPM moves the emulator into the confidential guest context and isolates it at VMPL0. The ordinary guest reaches it through a standard Linux TPM driver, while the host is used only to arrange execution. The AMD vTPM protocol follows the command framing of the official TPM 2.0 reference implementation's simulator protocol and defines query and command calls [4].

2.6.3 Linux TPM interface

Linux exposes TPMs through `/dev/tpm0` and, when the resource manager is available, `/dev/tpmrm0`. Userspace tools can use the resource-managed device without knowing whether the underlying implementation is a physical TPM, a host vTPM, or an SVSM vTPM. The `tpm2-tools` project provides commands for capabilities, PCR operations, object creation, policy sessions, and unsealing [12, 13].

This interface stability is central to Coconut-SVSM's practical value. A specialized secure service is most useful when existing applications can consume it without a new userspace API. The architecture chapter shows how Linux preserves that interface while translating requests into SVSM calls.

2.7 Security boundary and terminology

Table 2.1 summarizes the trust assumptions used throughout this thesis.

Table 2.1: Security boundary used in the thesis.

Component	Role in model	Interpretation
AMD CPU and secure firmware	Trusted foundation	Enforces memory encryption, RMP ownership and permissions, protected VMSAs, and attestation keys.
Coconut-SVSM at VMPL0	Trusted service layer	Holds vTPM code and state and mediates operations unavailable to the lower-privilege guest.
Guest Linux at VMPL2	Not trusted by vTPM	May request service but should not directly read or alter VMPL0-private state. It remains trusted by guest applications for normal OS behavior.
Host Linux, KVM, and QEMU	Untrusted for confidentiality and integrity	Allocate resources and schedule transitions but should not access private guest or vTPM state. They remain able to deny service.
Guest application	Workload-dependent	Uses standard TPM APIs and may rely on the TPM policy result.

In this thesis, functional maturity refers to representative operations working through the ordinary Linux interface. Standards certification, complete feature coverage, deployment hardening, and adversarial resilience are broader questions. The term *overhead* is used when a comparison isolates a baseline; other measurements are described as observed costs or durations.

Chapter 3

Coconut-SVSM Architecture and Linux Integration

3.1 Complete evaluated stack

Coconut-SVSM is a complex system that spans six independently built components: host Linux and KVM, QEMU, the IGVM library and image, EDK2/OVMF, Coconut-SVSM, and guest Linux. Figure 3.1 shows their roles.

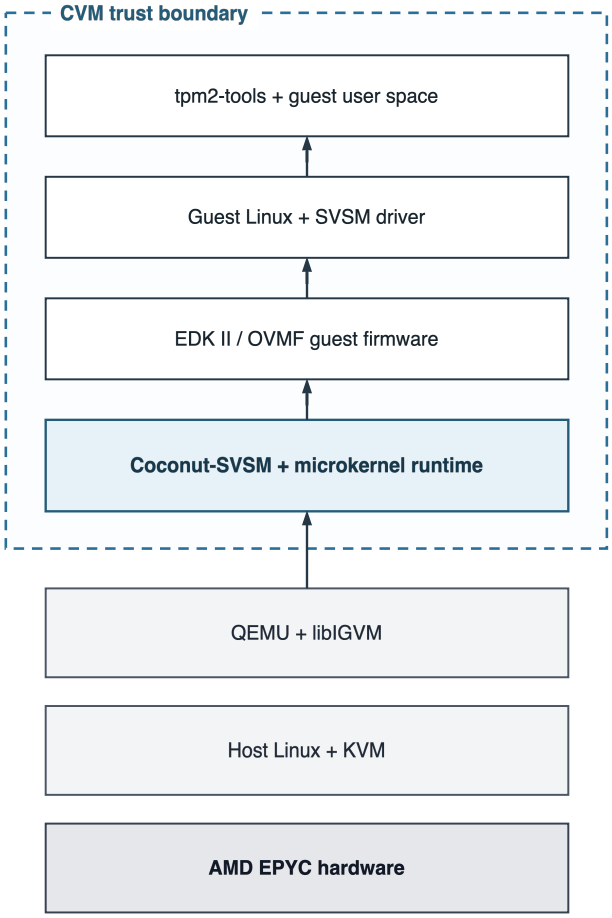

Figure 3.1: Components in the evaluated Coconut-SVSM stack. The host remains necessary for construction and scheduling but is outside the confidential trust boundary.

The exact version of every source component matters because the interfaces are evolving together. The Coconut installation documentation provides compatible branches rather than

assuming that any upstream kernel and QEMU release can be combined [11]. This thesis records complete commit hashes in Chapter 4 and Appendix A.

3.2 Launch and privilege establishment

3.2.1 Normal and SNP configurations

The normal VM uses QEMU's `q35` machine with KVM acceleration, an `EPYC-v4` CPU model, eight virtual CPUs, and 8 GiB of memory. OVMF is supplied through `-bios`. The guest disk and cloud-init seed are attached through virtual devices.

The SNP baseline uses the same high-level guest configuration but adds a private memory backend and a `sev-snp-guest` object. QEMU and KVM cooperate with the AMD Secure Processor to launch protected pages according to the SNP firmware ABI [7]. OVMF is still supplied directly through `-bios`.

3.2.2 SNP with Coconut-SVSM

The SVSM configuration adds an `igvm-cfg` object and supplies `coconut-qemu.igvm`. That artifact contains both the SVSM and OVMF images in the evaluated build. The secure launch therefore begins at VMPL0. Coconut-SVSM initializes its private memory and per-CPU state, then launches the firmware at the lower guest VMPL. Linux later discovers the SVSM fields placed in the secrets page.

This difference explains why "SNP versus SNP with SVSM" should be understood as a system-configuration comparison. It includes the way firmware is loaded and measured. The comparison is still operationally meaningful because these are the commands an operator would use to launch the two stacks, but it does not isolate a single function or binary.

3.3 Linux discovery and device registration

The guest kernel enables the AMD memory-encryption, TPM, and SVSM TPM configuration options. During early boot, the SEV code reads the secrets page and records the guest VMPL and SVSM Calling Area. Linux reserves the SVSM region so that the page allocator does not reuse it. Once the platform device is registered, the `tpm_svsm` driver probes the service. The exact option names are recorded in Chapter 4.

The upstream driver allocates a page-sized, physically contiguous request buffer, creates a TPM chip object, probes whether it is a TPM 2.0 device, and registers it with the TPM subsystem [14]. The evaluated guest then exposes `/dev/tpm0` and `/dev/tpmrm0`. This is the first important compatibility boundary: applications see the ordinary Linux device model.

The SVSM vTPM driver entered the upstream Linux kernel in the Linux 6.16 development cycle [11]. The experiment uses a Linux 7.0 Ubuntu guest, but that version is the evaluated version rather than the first possible version with upstream vTPM support.

3.4 End-to-end vTPM command path

Figure 3.2 summarizes the path established from the source investigation. Each box represents a boundary at which the request changes representation or protection context.

3.4.1 Userspace to `tpm_svsm_send`

The userspace tools do not issue SVSM-specific calls. They submit a binary TPM command to the Linux TPM resource manager. The TPM class eventually calls the `send` callback of the registered chip. For this device, that callback is `tpm_svsm_send()`.

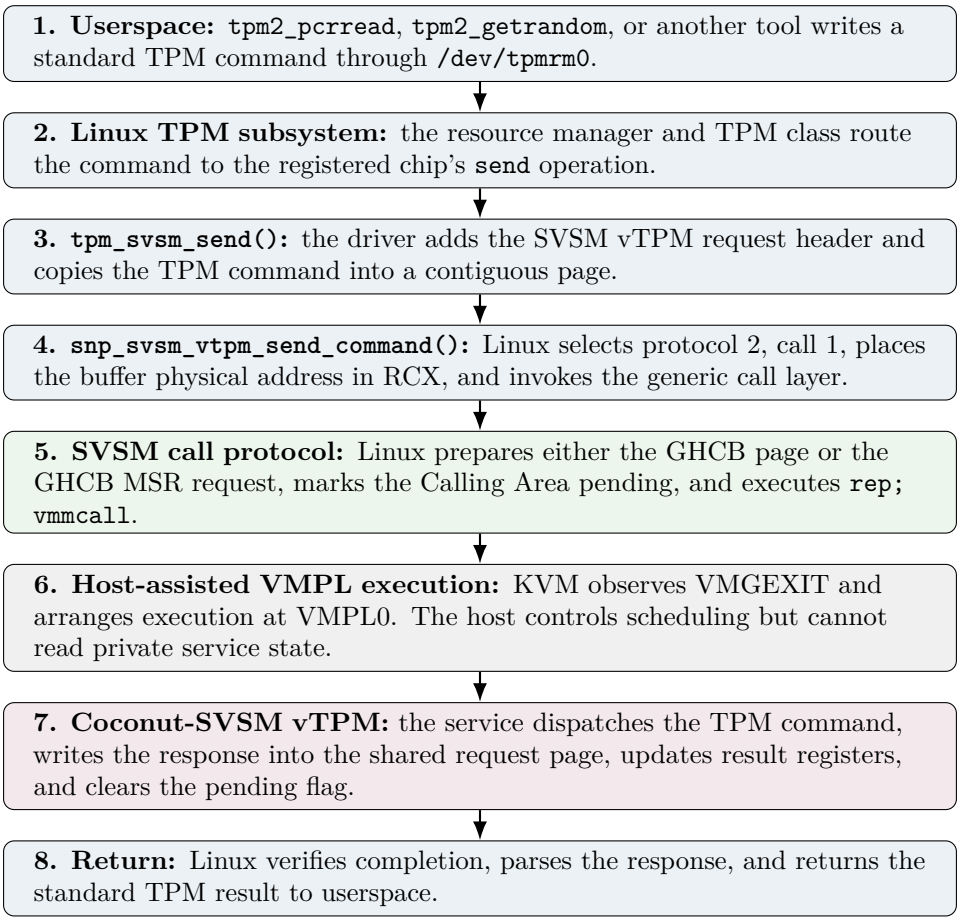

Figure 3.2: Source-guided vTPM command path. The host-side dispatch is shown at architectural granularity because this thesis did not establish every KVM function on that segment.

The driver first calls `svsm_vtpm_cmd_request_fill()`. That helper sets the platform command to `TPM_SEND_COMMAND`, records locality zero, stores the TPM command length, and copies the command bytes after the SVSM header. The copy is necessary because the SVSM protocol requires the header and TPM payload to occupy one physically contiguous request buffer.

The same page later holds the response. After the SVSM returns, `svsm_vtpm_cmd_response_parse()` validates the response size and copies the standard TPM response into the caller's buffer. The driver therefore acts as a narrow adapter. The TPM semantics remain in the Linux TPM subsystem and the vTPM implementation, while the driver translates transport framing.

3.4.2 Building the generic SVSM request

The architecture-specific helper `snp_svsm_vtpm_send_command()` constructs a generic `svsm_call`. RAX is set to the combined identifier for vTPM protocol 2 and command call 1. RCX receives the guest-physical address of the contiguous request page. The helper then invokes `svsm_perform_call_protocol()`.

This layering is useful because the same generic call mechanism serves protocols other than the vTPM. Core operations such as page validation and vCPU creation use protocol 0; Attestation uses protocol 1; and the vTPM uses protocol 2. The calling convention remains stable while each protocol defines the meaning of the argument registers and memory.

3.4.3 GHCB selection and interrupt control

`svsm_perform_call_protocol()` saves local interrupt state and chooses a transport. Once per-CPU GHCBs are initialized, it obtains the current CPU's GHCB. During earlier stages it may use a boot GHCB, and if no GHCB page is available it falls back to the MSR protocol.

For the page-based GHCB path, `svsm_perform_ghcb_protocol()` invalidates old GHCB state, sets the negotiated GHCB version, records `SVM_VMGEXIT_SNP_RUN_VMPL`, and exposes the GHCB physical address through the GHCB MSR. It then calls `svsm_issue_call()`.

The call layer retries when the SVSM returns an incomplete or busy result. Interrupts are restored after the operation. This means the measured `svsm_perform_ghcb_protocol()` region includes more than the instruction that changes execution context.

3.4.4 The guest-side boundary

The deepest guest-side function identified in this study is shown in Listing 3.1. The listing is shortened to the operations needed for the explanation.

Listing 3.1: Essential structure of the Linux guest-side SVSM issue function.

```
void svsm_issue_call(struct svsm_call *call, u8 *pending)
{
    register unsigned long rax asm("rax") = call->rax;
    register unsigned long rcx asm("rcx") = call->rcx;
    register unsigned long rdx asm("rdx") = call->rdx;
    register unsigned long r8  asm("r8")  = call->r8;
    register unsigned long r9  asm("r9")  = call->r9;

    call->caa->call_pending = 1;

    asm volatile("rep; vmmcall\n\t"
                 : "+r" (rax), "+r" (rcx), "+r" (rdx),
                   "+r" (r8), "+r" (r9)
                 : : "memory");

    *pending = xchg(&call->caa->call_pending, *pending);
```

```
17
18      call->rax_out = rax;
19      call->rcx_out = rcx;
20      call->rdx_out = rdx;
21      call->r8_out  = r8;
22      call->r9_out  = r9;
23  }
```

The fixed register variables instruct the compiler to place values in the registers defined by the ABI. The inline assembly already contains the mnemonic for the instruction sequence; the compiler and assembler do not decide where it should execute. The processor treats `rep; vmmcall` as VMGEXIT in this SEV context, returning control to the virtualization layer.

The `xchg` is atomic. If the returned old value is zero, the SVSM cleared the request and Linux can examine RAX for the result. If the value is still one, the host returned without causing the requested service execution. Linux then rejects or retries the operation rather than trusting the return registers.

3.5 Host mediation and the boundary of the trace

It is tempting to draw the boundary as a direct call from the guest kernel to Coconut-SVSM. The source and specification show a more careful model. VMGEXIT first transfers control to the host virtualization stack. The host observes the request to run VMPL0 and selects the appropriate VMSA. The SVSM then reads the calling vCPU's lower-VMPL VMSA, validates the exit boundary and pending flag, processes the request, and prepares the return.

The AMD SVSM specification intentionally does not standardize the mechanism by which a particular VMM loads or invokes the SVSM [4]. The guest-side trace in this thesis establishes the request down to VMGEXIT. It also identifies the architectural purpose of KVM's VMPL dispatch. This boundary is important when interpreting the latency measurements because the measured guest function includes the host-mediated interval but does not separate its internal components.

3.6 Why the call path matters

The path supports three conclusions that are independent of the performance results.

First, Linux preserves application compatibility. Existing tools use the normal TPM character device, and the SVSM-specific logic is contained below the TPM class interface.

Second, the untrusted host is still operationally necessary. Confidential computing removes it from the confidentiality and integrity trust boundary; it does not remove it from scheduling or availability. A malicious host may refuse to run VMPL0, and the pending handshake allows the guest to detect that failure.

Third, the protocol is deliberately reusable. The same call structure supports vTPM, attestation, page validation, and vCPU lifecycle operations. The performance of the boundary can therefore matter beyond the specific TPM service tested in this thesis, although the reported command counts and latencies should not be generalized to unmeasured protocols.

Chapter 4

Experimental Methodology

4.1 Study design

The evaluation combines quantitative measurement, functional testing, and source inspection shown below.

- Boot profiling measures a deployment-level outcome: how long the complete launch configurations take to reach an explicit guest-ready marker, and where time is spent across visible phases.
- Function tracing measures a narrow runtime region: the duration of the guest kernel's GHCB-based SVSM call protocol for representative vTPM commands.
- Boot memory logs measure the guest kernel's visible memory accounting, instead of the total private memory allocated to the paravisor.
- The vTPM workflow tests command interoperability and policy semantics through standard Linux tools.
- Source inspection explains how the observed command crosses the guest-to-SVSM boundary and identifies what the timing region includes.

The three launch configurations are shown in Table 4.1. The normal VM separates the general cost of confidential execution from ordinary virtualization. The SNP baseline separates the additional deployment-level behavior associated with Coconut-SVSM from SNP alone.

Table 4.1: Compared virtual-machine configurations.

Name	Confidential protection	Firmware loading	Privileged service layer
Normal	None	OVMF through `-bios`	None
SNP	AMD SEV-SNP	OVMF through `-bios`	None
SNP with SVSM	AMD SEV-SNP	OVMF embedded in IGVM	Coconut-SVSM at VMPL0

4.2 Hardware and host software

All experiments were performed on one Supermicro server. Table 4.2 records the relevant hardware and host software.

Table 4.2: Host experimental environment.

Component	Value
Server	Supermicro Super Server
Processor	AMD EPYC 9175F
CPU topology	One socket, 16 physical cores, one hardware thread per core
Host memory	256 GiB DDR5 ECC, configured at 6000 MT/s
Host operating system	Ubuntu 24.04.4 LTS
Host kernel	Linux 6.11.0-based Coconut-SVSM branch
Host kernel commit	`3d7f4e43fb9b1b41f1f0ef09c8f6d770505e59ce`
Hypervisor	In-kernel KVM with `kvm_amd`
SEV firmware	API 1.58, build 6
SEV-SNP status	Enabled successfully by `kvm_amd`; SNP ASIDs 1 through 63 available

The server contains substantially more host memory and physical CPU capacity than the guest requires. There was no attempt made to generalize performance across processors or firmware versions and the purpose of the detailed configuration is scope and reproducibility.

4.3 Coconut-SVSM software stack

Table 4.3 records the exact repositories and commits because each component crossed different development branches.

Table 4.3: Source repositories and commits used to construct the stack.

Component	Repository and branch	Commit
Host Linux	`coconut-svsm/linux`, `svsm`	`3d7f4e43fb9b1b41f1f0ef09c8f6d770505e59ce`
QEMU	`coconut-svsm/qemu`, `svsm-igvm`	`34d36846c6bc670bdb632b94be11d386d4c925d0`
IGVM library	`microsoft/igvm`, `main`	`fc0fa3f55cde54e016a1169926ae39f9561aa8f7`
EDK2/OVMF	`coconut-svsm/edk2`, `svsm`	`5035a8789f556bd189a22c3aa594bb9048111461`
Coconut-SVSM	`coconut-svsm/svsm`, `main`	`44f639d2efbe481c5ba15a152571da90e5271cb0`

The build environment used Rust and Cargo 1.88.0, LLVM 20.1.5, GCC 13.3.0, and GNU Binutils 2.42. The resulting `coconut-qemu.igvm` artifact was 4,831,344 bytes and had SHA-256 digest `d65d90c850f424f46241b33b47bc47798c2dc51e532ab8d96419bc14e0b8b9d0`. The artifact size is recorded for integrity and reproducibility.

4.4 Guest configuration

The guest ran Ubuntu 26.04 LTS on x86-64. The installed kernel was Linux 7.0.0-22-generic, built with `CONFIG_AMD_MEM_ENCRYPT=y`, `CONFIG_TCG_TPM=y`, and `CONFIG_TCG_SVSM=m`. All 30 final boot logs and the vTPM functionality record identify this kernel version.

In the SVSM guest, kernel logs reported AMD SEV, SEV-ES, and SEV-SNP as active; the guest ran at VMPL2; the SVSM vTPM 2.0 device initialized; and the SEV guest driver used the VMPCK2 communication key. The platform driver appeared under `/sys/bus/platform/driv`

Table 4.4: Virtual-machine configuration common to the comparisons.

Setting	Value
Virtual CPU model	AMD EPYC-v4
Virtual CPUs	8: one socket, eight cores, one thread per core
Configured memory	8 GiB
Guest disk	10 GB QEMU QCOW2 disk, ext4 root filesystem
Firmware	UEFI through OVMF
Network	Intel E1000 emulation with user-mode networking
Storage controller	Virtio SCSI
Display	Disabled
Serial output	QEMU serial pipe for timing runs

`ers/tpm-svsm`, the `tpm_svsm` module was loaded, and both `/dev/tpm0` and `/dev/tpmrm0` were present.

4.5 Experiment 1: VM cold-start profiling

4.5.1 Metric and markers

The boot metric begins immediately before starting QEMU and ends when the guest emits the explicit string `CVM ready`. The ready marker is preferable to a login prompt because it can be produced deterministically after the intended userspace initialization. Each serial line receives a timestamp relative to the launch start.

The timeline analysis uses stable text markers to divide visible boot activity:

Table 4.5: Serial markers used for boot-phase localization.

Phase	Start marker	End marker
Launch to BDS	QEMU launch at $t = 0$	`BdsDxe: starting Boot0003`
BDS to Linux	`BdsDxe: starting Boot0003`	Linux 7.0.0-22 version banner
Linux to systemd	Linux 7.0.0-22 version banner	First systemd system-mode banner
systemd to ready	First systemd system-mode banner	`CVM ready`

The marker boundaries are observable software milestones, not perfect hardware phase boundaries. For example, the interval from `BdsDxe` to the Linux banner includes bootloader activity and any serial work emitted in that interval. It should not be equated with vTPM execution alone.

4.5.2 Repeated measurement protocol

The final dataset contains 10 independent QEMU launches for each configuration, for 30 launches in total. The runs were collected in three configuration blocks: SVSM, SNP, and normal VM. Each run used the same guest image and the launch scripts' `snapshot=off` disk policy. The logs preserve output beyond the ready marker because the guest was shut down afterward, but only the first `CVM_READY` record is used as the endpoint.

A single long-lived reader timestamped serial lines relative to launch. This replaced the earlier shell loop that spawned `date`, `awk`, and `grep` for every line. Avoiding per-line process creation reduces observer overhead, but it does not remove the cost of firmware debug output itself. QEMU and firmware can still be delayed by a high-volume serial path, so serial volume is analyzed alongside elapsed time.

All 30 runs reached the endpoint and are retained. The primary statistic is the median ready time because it is robust to occasional launch stalls. The interquartile range reports typical spread, and every point remains visible in the distribution figure. A secondary marker analysis localizes differences without assigning them to one component. Quartiles were calculated using NumPy's linear percentile interpolation. The complete parser and derived CSV files are included with the thesis source.

4.6 Experiment 2: SVSM protocol-call latency

Linux ftrace was used to time `svsm_perform_ghcb_protocol()` [15]. Three standard commands were selected:

- `tpm2_pcrread sha256:0`, representing a small state read;
- `tpm2_getrandom 32`, representing random-byte generation; and
- `tpm2_getcap properties-fixed`, representing a capability query.

Each command was executed 50 times. The trace was grouped by userspace invocation so that two quantities could be calculated: the duration of each function call and the sum of all traced calls caused by one command. A separate userspace timing provided the approximate 30 ms end-to-end command duration.

The unit of interpretation is crucial. The traced function includes GHCB preparation, Calling Area manipulation, VMGEXIT, host-mediated VMPL0 execution, SVSM processing, return, and result checking. It is therefore reported as *guest-observed end-to-end SVSM protocol-call latency*, not the *latency of a hardware VMPL switch*.

4.7 Experiment 3: Linux-visible memory accounting

The memory experiment compared the kernel `Memory:` line between SNP and SNP with SVSM. Three observations were recorded for each configuration. The line reports available, total, and reserved memory after the kernel has processed firmware memory maps and its own reservations.

The analysis calculates:

$$\Delta_{\text{total}} = M_{\text{total,SNP}} - M_{\text{total,SVSM}}, \tag{4.1}$$

and the difference between the mean reported reserved values. Each difference is divided by the configured 8 GiB guest size to express its operational scale.

4.8 Experiment 4: vTPM functional workflow

The functional evaluation uses unmodified `tpm2-tools`. Table 4.6 groups the commands by capability rather than presenting a long terminal transcript in the main text.

The positive and negative policy cases are both necessary. A successful unseal shows that the command path and object state work. Failure after a PCR change shows that the result depends on measured state.

The record contains the commands, TPM output, PCR values, independently calculated extension values, successful byte comparisons, and the TPM error returned by the changed-PCR test. Exact evidence is summarized in Chapter 6, and the command sequence is reproduced in Appendix B.

Table 4.6: vTPM functional test groups.

Test group	Representative commands	Success condition
Discovery and basic commands	`ls /dev/tpm*`, `tpm2_getcap`, `tpm2_getrandom`, `tpm2_pcrread`	Device is registered and commands return valid output.
PCR state transition	`tpm2_pcrreset`, `tpm2_pcrextend`, `tpm2_pcrread`	PCR 16 resets and changes after extension.
Object lifecycle	`tpm2_createprimary`, `tpm2_create`, `tpm2_load`	Primary and sealed object are created and loaded.
Basic sealing	`tpm2_unseal`, `cmp`	Recovered bytes exactly match the original secret.
PCR-policy success	`tpm2_policypcr`, policy session, `tpm2_unseal`	Secret is released while PCR 16 matches the policy input.
PCR-policy rejection	Extend PCR 16, repeat old policy	Old policy cannot be satisfied and the secret is not released.

4.9 Source-level tracing method

The code investigation began at the userspace-visible TPM device and followed function calls through the Linux TPM driver. Searches were performed in the guest kernel for `tpm_svsm_send`, `snp_svsm_vtpm_send_command`, `svsm_perform_call_protocol`, and `svsm_issue_call`. The lowest guest-side boundary was identified by the inline `rep; vmmcall` sequence.

The analysis distinguishes observed source from inferred host behavior. The Linux path and AMD calling convention are directly supported by source and specification. The statement that KVM arranges VMPL0 execution is part of the architecture. The exact function-by-function KVM-to-SVSM dispatch path was not completed and is therefore not presented as a traced result.

Chapter 5

Operational Cost Evaluation

5.1 Overview

This chapter evaluates three different notions of cost. Boot time is a macro-level deployment cost. SVSM protocol latency is a micro-level cost paid when a service is invoked. Linux-visible memory accounting is a capacity cost. Their magnitudes cannot be compared directly, but together they reveal where Coconut-SVSM is visible to an operator and where it is not.

5.2 VM cold-start behavior

5.2.1 Repeated-run result

Table 5.1 reports the final repeated-run results. Each cell contains the median and interquartile range across 10 launches.

Table 5.1: Repeated VM cold-start time to the `CVM ready` marker.

Configuration	Runs	Median	IQR	Range
Normal VM	10	9.386 s	0.108 s	9.258–9.457 s
SEV-SNP	10	14.516 s	0.150 s	14.251–63.803 s
SEV-SNP with Coconut-SVSM	10	20.504 s	0.228 s	20.210–21.512 s

Relative to the normal VM, SNP increased the median by 5.130 s (54.7%). Adding Coconut-SVSM to SNP increased the median by a further 5.988 s (41.3%). The complete SVSM configuration was therefore 11.119 s (118.5%) slower than the normal VM at the selected endpoint.

5.2.2 The SNP outlier

Nine SNP runs fell between 14.251 s and 14.621 s. One run reached the marker at 63.803 s. Its trace contains a 50.313 s gap after systemd started `initrd-switch-root.service` and before the next serial output. The firmware and early Linux markers before that point align with the other SNP runs. The outlier is therefore localized to a transient switch-root stall rather than the SNP launch path as a whole.

The run is retained because the guest reached the valid endpoint and no launcher failure was recorded. It illustrates why the median is the primary statistic: the SNP mean rises to 19.391 s with a sample standard deviation of 15.606 s, while the median remains 14.516 s with an IQR of only 0.150 s.

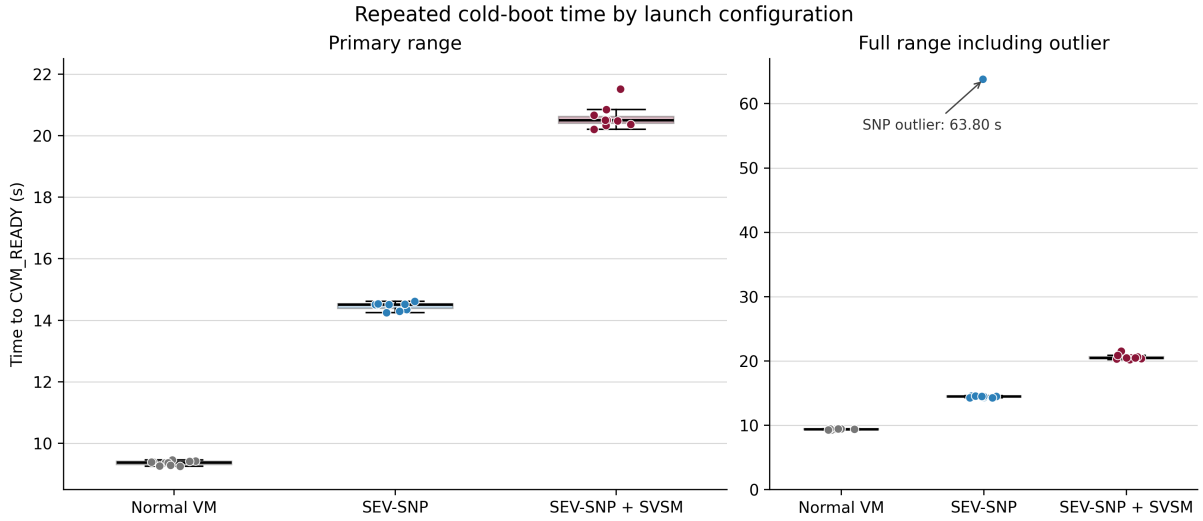

Figure 5.1: Repeated cold-start observations. The left panel enlarges the primary range; the right panel retains the 63.803 s SNP observation. Boxes show the interquartile range and center lines show medians. All runs are included.

5.2.3 Repeated phase localization

Figure 5.2 compares median cumulative marker timestamps. The phase data reproduce the earlier qualitative observation with repeated evidence: the dominant SNP-to-SVSM difference is before the Linux banner.

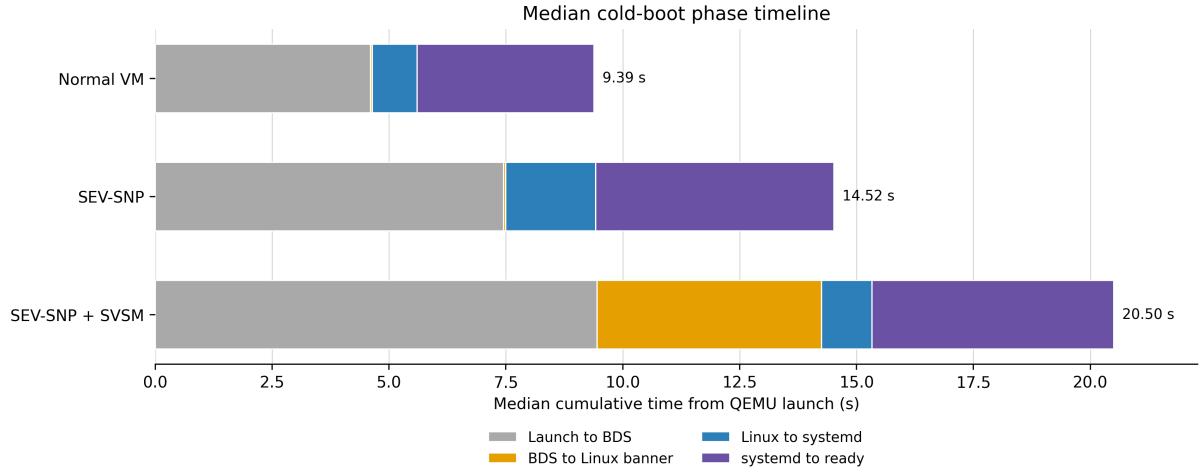

Figure 5.2: Median cumulative cold-boot timeline. Each boundary is positioned at the median timestamp of the corresponding marker. The BDS-to-Linux interval expands only in the SVSM configuration.

The median BDS-to-Linux duration was 0.036 s for the normal VM, 0.047 s for SNP, and 4.793 s for SNP with SVSM. The SVSM interval therefore added approximately 4.746 s relative to the SNP median phase duration. This interval alone is approximately 79.3% of the 5.988 s median endpoint difference.

The line counts change how this interval should be interpreted. Normal and SNP each emitted a median of 18 timestamped lines between the two markers. SVSM emitted a median of 2,106 lines. The additional output contains PCR reads, TPM event-log entries, hash digests, and other measured-boot debug information. The roughly 2,088 extra lines coincide with almost the entire added interval.

This close alignment does not prove that serial output explains every microsecond, but it rules

Table 5.2: BDS-to-Linux interval and serial volume. Values are medians across ten runs.

Configuration	Interval duration	Timestamped lines
Normal VM	0.036 s	18
SEV-SNP	0.047 s	18
SEV-SNP with Coconut-SVSM	4.793 s	2106

out a clean attribution to vTPM cryptography or VMPL switching. The repeated experiment measures the launch time of the complete debug-instrumented stack. A release-firmware or serial-disabled ablation is required to estimate production-oriented startup cost.

5.2.4 Operational meaning

Boot cost matters differently across deployment models. A long-lived database or service amortizes several seconds of startup over hours or days. A serverless platform, autoscaling tier, short batch job, or repeatedly created sandbox may pay the cost frequently. Recent CVM measurements similarly identify boot and context-transition behavior as important system-level dimensions [16].

For the evaluated stack, the median SVSM endpoint cost is substantial enough to matter for short-lived or rapidly autoscaled guests. However, the phase and line-volume evidence also shows that this number should not be carried directly into a production capacity model. The correct next experiment is an ablation that keeps the SVSM configuration constant while comparing debug and release firmware, suppressing event-log dumping, or disabling vTPM-related measurement. Until then, the result is a reproducible complete-stack observation rather than an intrinsic paravisor cost.

Boot finding

Across ten launches per configuration, median time to `CVM ready` was 9.386 s for the normal VM, 14.516 s for SNP, and 20.504 s for SNP with SVSM. The SVSM configuration added 5.988 s over SNP. Approximately four-fifths of that median difference appeared between BDS start and the Linux banner, where the SVSM logs emitted about 2,100 timestamped debug lines. The result characterizes the evaluated debug stack and does not isolate an inherent SVSM or vTPM penalty.

5.3 SVSM protocol-call latency

5.3.1 Results

Table 5.3 reports the retained summary across 50 executions of each command. A single userspace command triggered multiple SVSM calls.

Table 5.3: Observed `svsm_perform_ghcb_protocol()` latency.

Command	Runs	Calls/run	Median call	Median aggregate/command
`tpm2_pcrread sha256:0`	50	3	35.9 µs	134 µs
`tpm2_getrandom 32`	50	3	45.8 µs	147 µs
`tpm2_getcap properties-fixed`	50	2	49.1 µs	112 µs

Median per-call latency ranged from 35.9 µs to 49.1 µs. The range across commands was therefore 13.2 µs. Aggregate traced time remained between 112 µs and 147 µs, even though two commands issued three calls and one issued two.

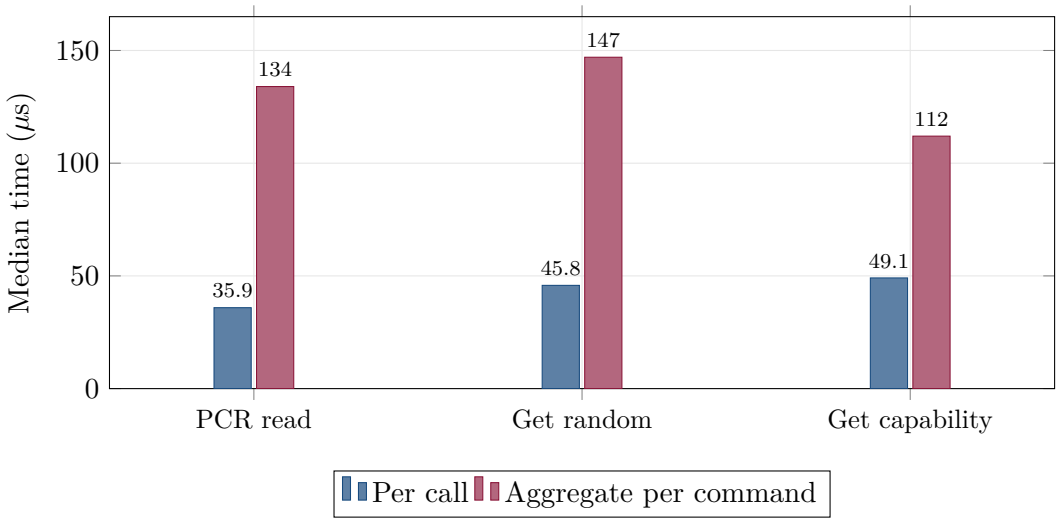

Figure 5.3: Median observed SVSM protocol time for the three tested commands. Aggregate time is the sum of the calls triggered by one userspace command.

5.3.2 Share of userspace command time

The approximate userspace command time was 30 ms, or 30,000 µs. The median aggregate protocol share is:

$$\text{PCR read} : \frac{134}{30000} \times 100 \approx 0.447\%, \tag{5.1}$$

$$\text{getrandom} : \frac{147}{30000} \times 100 \approx 0.490\%, \tag{5.2}$$

$$\text{getcap} : \frac{112}{30000} \times 100 \approx 0.373\%. \tag{5.3}$$

For the tested commands, the traced SVSM region accounted for less than approximately 0.5% of the observed command time. Establishing overhead requires running the same command against a comparable non-SVSM TPM and subtracting the baseline. The current result instead establishes that this particular guest-side protocol region is not the dominant component of the command's absolute latency.

5.3.3 Why commands issue multiple calls

The number of SVSM transactions depends on the interaction between the TPM core and device driver. A userspace operation may require status checks, command submission, and response handling rather than one monolithic transition. The observed counts show that command-level cost cannot be estimated by multiplying one universal VMPL-switch number by one.

The capability command had the highest median per-call value but the lowest aggregate because it issued two calls. PCR read had the lowest per-call median but a larger aggregate. This illustrates why both metrics are needed.

5.3.4 What the timing includes

The measured function includes:

1. invalidating and filling the GHCB;
2. writing the GHCB address to the MSR;

3. preparing the SVSM Calling Area and fixed registers;
4. executing VMGEXIT;
5. host processing needed to run the VMPL0 context;
6. SVSM validation and service dispatch;
7. vTPM-side processing reached by that call;
8. return to the lower VMPL;
9. checking the pending field, exception state, and result code.

It may also include interference from interrupts, scheduling, translation lookaside buffer effects, interprocessor synchronization, or other implementation work. The experiment cannot apportion the 35.9 µs to 49.1 µs among these components.

A deeper attribution study would require timestamps in multiple execution contexts: before VMGEXIT in the guest, inside KVM's VMGEXIT handler, at VMPL0 entry, around service dispatch, at SVSM return, and after guest reentry. Those clocks would then need to be synchronized or expressed in a consistent cycle domain. Such a harness is valuable but substantially broader than function tracing.

Latency finding

Across three representative commands, a guest-observed GHCB-based SVSM call took tens of microseconds. Multiple calls contributed a median aggregate of 112 µs to 147 µs per command, below approximately 0.5% of the observed 30 ms command time.

5.4 Linux-visible memory accounting

5.4.1 Raw observations

Table 5.4 shows the retained kernel memory lines in structured form. Total memory was stable within each configuration. Reserved memory varied by only several kilobytes.

Table 5.4: Linux boot memory observations in KiB.

Configuration	Observation	Available	Total / Reserved
SNP	1	7,534,448	8,382,640 / 837,748
SNP	2	7,534,624	8,382,640 / 837,744
SNP	3	7,534,664	8,382,640 / 837,748
SNP with SVSM	1	7,534,424	8,382,612 / 838,000
SNP with SVSM	2	7,534,392	8,382,612 / 837,996
SNP with SVSM	3	7,534,124	8,382,612 / 838,000

Linux reported 28 KiB less total memory with SVSM:

$$8{,}382{,}640 - 8{,}382{,}612 = 28 \text{ KiB}. \tag{5.4}$$

Mean reserved memory was 837,746.7 KiB for SNP and 837,998.7 KiB for SNP with SVSM, a difference of approximately 252 KiB.

5.4.2 Relative scale

Relative to the configured 8 GiB, the total-memory difference is:

$$\frac{28}{8 \times 1024 \times 1024} \times 100 \approx 0.000334\%. \tag{5.5}$$

The reserved-memory difference is:

$$\frac{252}{8 \times 1024 \times 1024} \times 100 \approx 0.00300\%. \tag{5.6}$$

Both are below 0.004%. At this scale, the difference does not meaningfully reduce the workload capacity of an 8 GiB guest.

5.4.3 Interpretation

The stable 28 KiB total difference is smaller than the multi-megabyte IGVM artifact and much smaller than any plausible complete runtime environment. This apparent mismatch is expected because Linux reports memory from its own view. Pages that are never exposed to the lower VMPL may not appear as an ordinary kernel reservation. Other pages may be represented indirectly through firmware memory maps.

The reserved category also aggregates many causes. It is not possible to assign all 252 KiB to Coconut-SVSM code or data. The result measures the effect visible to the Linux allocator after the full stack is constructed.

A complete resident-footprint measurement would require the IGVM memory map, SVSM linker layout, runtime allocator state, per-vCPU structures, service heaps, shared pages, and any memory deposited by the guest. That measurement remains future work.

Memory finding

Coconut-SVSM changed Linux-visible memory accounting by less than 0.004% of the configured guest memory. The capacity effect is negligible in the evaluated 8 GiB VM, but the measurement does not reveal the complete VMPL0-resident footprint.

5.5 Cross-metric synthesis

The cost results do not show a uniform “SVSM overhead.” Instead they separate into distinct regimes:

- The measured steady-state protocol region is short compared with the tested TPM command.
- The Linux-visible memory effect is extremely small relative to guest capacity.
- The boot trace contains the largest visible difference and the greatest methodological sensitivity.

This pattern suggests that Coconut-SVSM’s practical cost in the evaluated stack is dominated by construction and pre-kernel integration rather than by the two steady-state quantities measured here. That conclusion remains scoped to the selected vTPM commands and guest size. High-frequency SVSM services, concurrent vCPUs, larger payloads, or a different firmware build may produce a different balance.

Chapter 6

vTPM Functional Evaluation

6.1 Evaluation objective

The purpose of this experiment is to determine whether the Coconut-SVSM vTPM behaves as a useful TPM 2.0 device through the ordinary Linux software stack. The test is deliberately broader than a device-enumeration check. A TPM can answer a capability query while still failing stateful object or authorization operations. The workflow therefore progresses from basic commands to a PCR-bound secret and includes a negative policy case.

TPM 2.0 defines a large command, algorithm, hierarchy, persistence, authorization, attestation, and error-handling space [1]. The selected operations exercise a representative path with direct relevance to measured boot and local secret release.

6.2 Device discovery and standard interface

The guest exposed two device nodes:

- `/dev/tpm0`, the direct TPM character device; and
- `/dev/tpmrm0`, the Linux resource-managed TPM device.

Kernel logs identified an `SNP SVSM vTPM 2.0 device`, the `tpm_svsm` module was loaded, and the driver appeared under `/sys/bus/platform/drivers/tpm-svsm`. Standard `tpm2-tools` 5.7 commands used the default `tcti-device` interface without an SVSM-specific userspace library. This confirms the architectural role described in Chapter 3: `tpm_svsm` translates the transport below the stable Linux TPM interface.

Three initial commands completed:

- `tpm2_getcap properties-fixed`, which queried fixed TPM properties;
- `tpm2_getrandom 16`, which returned random bytes; and
- `tpm2_pcrread sha256:0,1,2,3,4,5,7`, which read selected SHA-256 PCRs.

The capability response identified TPM family 2.0, 24 PCRs, and 119 library commands. These values describe the interface exposed by the evaluated firmware TPM; they are not a conformance result. Together, the commands establish connectivity and basic TPM command handling.

6.3 PCR state transitions

PCR 16 was selected because it can be reset in the tested environment and is convenient for a controlled experiment. The sequence was:

1. read the initial PCR 16 value;
2. reset PCR 16;
3. hash the string `first measurement` with SHA-256;
4. extend PCR 16 with that digest; and
5. read PCR 16 again.

The captured initial or reset value was:

```
0x0000000000000000000000000000000000000000000000000000000000000000
```

After the first extension, PCR 16 was:

```
0xDBDFD380040992F64BBB35B17D3BE7B74A496B8B13D898844A9ED4906390C949
```

The measurement digest for `first measurement` was:

```
12e773f6a5bc5678e2af3284d997d181b2dc38136535c7f491aa9f97b4f3064d
```

Independently calculating SHA-256 over the old all-zero PCR concatenated with that digest produced:

```
dbdfd380040992f64bbb35b17d3be7b74a496b8b13d898844a9ed4906390c949
```

This exactly matched the vTPM output. It is stronger evidence than observing a changed value because it confirms the expected hash-chain transition defined by Equation 2.1.

6.4 Basic object creation and sealing

The next stage tested a simple TPM object lifecycle:

1. create an ECC primary object in the owner hierarchy;
2. create a sealed object containing the bytes `svsm-vtpm-secret`;
3. load the public and private object material under the primary;
4. unseal the object to a file; and
5. compare the recovered file byte-for-byte with the original.

The `cmp` command reported success. This is stronger than printing the returned text because it checks exact equality and exits nonzero on a mismatch. The result shows that the tested vTPM supported primary creation, sealed-object creation, object loading, and unsealing within one guest session.

6.5 PCR-bound policy

6.5.1 Policy construction

PCR 16 was reset and read into `pcr16.bin`. A trial authorization session then executed `tpm2_policypcr` with the SHA-256 PCR 16 selection, producing `pcr16.policy`. Both the command and the saved policy file contained:

```
bff2d58e9813f97cefc14f72ad8133bc7092d652b7c877959254af140c841f36
```

The digest was associated with a new sealed object containing `pcr-bound-svsm-secret`.

The policy construction follows the normal `tpm2-tools` flow [13]. It is significant that the procedure required no special accommodation for the SVSM transport.

6.5.2 Matching-state success

A policy session was started while PCR 16 still matched the saved state. `tpm2_policypcr` satisfied the session, and `tpm2_unseal` released the protected bytes. A second `cmp` verified equality with the original secret.

The positive case jointly exercises PCR state, policy-digest generation, a policy-bound object, session state, authorization, and unseal output. It shows that the individual commands compose into a useful workflow.

6.5.3 Changed-state rejection

The string `changed measurement` was hashed and extended into PCR 16. This produced a new PCR state. The same policy session sequence was then attempted using the old policy input.

The changed PCR 16 value was:

```
0x43DB69D614F6862F5FF258B5D359345325725C1D5BCB91BA53271E0461C7CC09
```

The policy command returned TPM error `0x1C4`, and the protected output was not produced.

Failure at `tpm2_policypcr` is expected. The policy session cannot be satisfied when the current PCR state differs from the state used to construct the policy. Since the authorization stage fails, the subsequent unseal is not authorized.

The digest of `changed measurement` was:

```
ccd9320e052c9a566a58e99c9c8a87a5048f758954b928dd9d70ef019e1724d2
```

Extending the all-zero PCR with this value independently produced:

```
43db69d614f6862f5ff258b5d359345325725c1d5bcb91ba53271e0461c7cc09
```

This again exactly matched the vTPM. The paired success and rejection therefore depend on a verified PCR-state change rather than an assumed one.

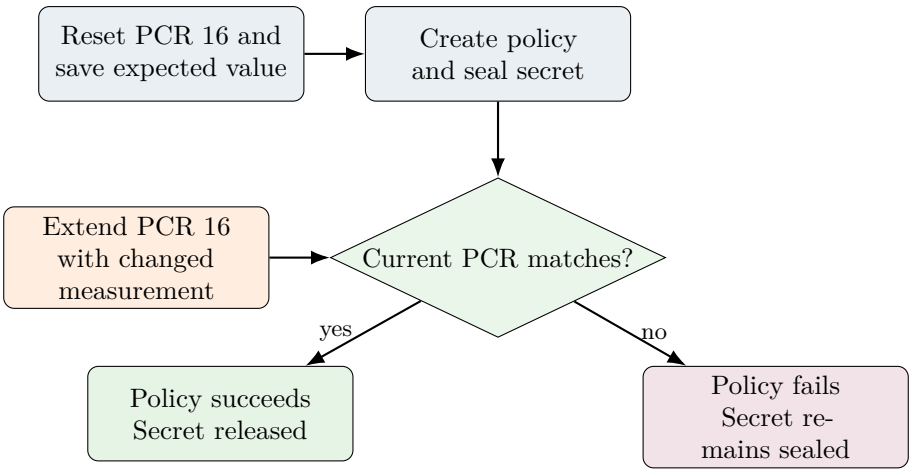

Figure 6.1: PCR-bound secret-release test. Both the successful and rejected paths were exercised.

6.6 Result summary

Table 6.1 summarizes the observed functional coverage.

Table 6.1: vTPM functional evaluation results.

Capability	Result	Evidence
Linux device discovery	Pass	`tpm_svsm` initialized; `/dev/tpm0` and `/dev/tpmrm0` present.
Fixed-property query	Pass	`tpm2_getcap properties-fixed` completed.
Random generation	Pass	`tpm2_getrandom 16` returned bytes.
PCR reads	Pass	Selected SHA-256 PCR banks were readable.
PCR reset and extend	Pass	PCR 16 reset and changed after extension.
Primary and sealed-object creation	Pass	Primary, public, private, and loaded context were created.
Basic unseal	Pass	Recovered bytes matched the original secret with `cmp`.
Matching PCR policy	Pass	Policy-bound secret was released when PCR 16 matched.
Changed-PCR policy	Pass	The old policy failed after PCR 16 changed; secret release was blocked.
Reboot persistence	Not tested	Object and vTPM state were not evaluated across VM recreation.
Remote attestation binding	Not tested	No relying party verified combined SNP and vTPM evidence.
Formal TPM conformance	Not tested	No official conformance suite was run.

6.7 Analysis

6.7.1 Why the negative case is the strongest functional evidence

Many device demonstrations stop after `getrandom` or a PCR read. Such tests prove that a command can reach a responder but say little about whether security decisions depend on protected state. The changed-PCR case is stronger because it checks a relationship:

$$\text{release} \iff \text{current PCR state satisfies sealed-object policy.} \tag{6.1}$$

The matching state produced release, while the changed state produced rejection. This paired outcome demonstrates the core mechanics of local measured-state authorization.

It still does not prove resistance to rollback or a malicious implementation. A defective vTPM could be written to produce the expected response for this specific sequence. The test establishes functional behavior in the evaluated system, while the architecture and hardware boundary provide the intended security argument.

6.7.2 Compatibility value

The experiment used `tpm2-tools` as documented for ordinary TPM devices. The guest application did not manage VMPLs, GHCB fields, or Calling Area state. This compatibility is a meaningful form of readiness because it reduces the changes needed above the kernel. Software that already uses the Linux TPM interface can potentially consume the service without knowing that the implementation lives at VMPL0.

6.7.3 Security meaning

The vTPM resides inside the SNP-protected guest context rather than in an ordinary host process. VMPL separation is intended to prevent the lower-privilege guest kernel from reading

or modifying VMPL0-private service state. This placement addresses the trust mismatch of a host-emulated vTPM.

The experiment does not independently verify every isolation property. It does not attack the host, attempt malicious RMP mappings, inspect service pages from the guest, or verify side-channel resistance. The functional result should therefore be read together with the documented SEV-SNP and SVSM trust model, not as a security proof.

6.8 Limits of functional maturity

Three missing dimensions are particularly important.

6.8.1 Persistence and rollback

The tested workflow occurs within one running guest. A practical vTPM may need to preserve endorsement identity, keys, nonvolatile indices, counters, and sealed objects across restarts. Saving that state outside the CVM reintroduces an untrusted storage problem. Encryption and integrity are necessary, but not sufficient, because a host could replay an older valid state or clone it into another instance.

Prior work on ephemeral SVSM vTPMs avoids persistent host state and focuses on remote attestation [17]. Coconut-SVSM development work also considers protected persistent state and rollback mitigation. Persistence is therefore not a minor untested command; it is a separate systems-security problem.

6.8.2 Attestation composition

The SNP hardware report can establish a measured guest launch. The vTPM can report PCR state and sign TPM evidence. A relying party ultimately needs confidence that the vTPM instance and its keys belong to the attested SNP guest and to the expected SVSM service. The AMD SVSM specification defines service-attestation data for this purpose [4], but the complete remote verification flow was not tested here.

6.8.3 Breadth and robustness

The test does not cover every algorithm, hierarchy, authorization method, locality, error condition, NV operation, or concurrent request pattern. It also does not measure recovery after a malformed command or service crash. These limitations prevent a claim of complete TPM 2.0 compliance or production robustness.

vTPM finding

After driver registration, the evaluated Coconut-SVSM vTPM supported standard Linux access, mutable PCR state, object creation, sealing, matching-policy unseal, and changed-PCR rejection. This is sufficient to demonstrate a representative local PCR-bound secret-release workflow.

Chapter 7

Discussion and Limitations

7.1 Answering the research questions

7.1.1 RQ1: observed operational costs

The measured costs are uneven across the lifecycle of the VM.

At runtime, the observed GHCB-based SVSM call path took tens of microseconds. The three selected userspace commands triggered two or three calls, but their median aggregate traced time remained between 112 µs and 147 µs. Against the approximate 30 ms command duration, that region accounted for less than about 0.5%. For these commands, the guest-observed protocol region was not the dominant latency component.

For memory, Linux saw a 28 KiB reduction in total memory and an approximately 252 KiB increase in reserved memory. Both were below 0.004% of the 8 GiB guest. The guest-visible capacity effect is therefore negligible in the evaluated configuration.

Boot behavior produced the largest visible configuration difference. Across ten launches per configuration, median time to the fixed ready marker increased from 9.386 s for the normal VM to 14.516 s for SNP and 20.504 s for SNP with SVSM. The 5.988 s SNP-to-SVSM difference was concentrated before Linux entry. The BDS-to-Linux interval expanded to a median 4.793 s and emitted approximately 2,100 timestamped lines, compared with 18 for SNP. The magnitude is therefore valid for the evaluated debug stack but cannot be interpreted as an isolated paravisor or vTPM cost.

7.1.2 RQ2: vTPM functional coverage

The vTPM completed a representative local security workflow through the standard Linux interface. It supported device discovery, capability and randomness commands, PCR reads, PCR reset and extension, object creation, basic sealing, a matching PCR policy, and rejection after the PCR changed.

The result is stronger than a smoke test because the negative policy case establishes state-dependent behavior and both PCR extensions independently matched the expected hash-chain calculation. The experiment covers representative functionality rather than a TPM conformance suite, leaving persistence, attestation composition, broader command coverage, and robustness for separate evaluation. The boot log also showed that IMA entered TPM-bypass before the device registered, which makes early device availability an important integration issue alongside the successful post-registration workflow.

7.1.3 RQ3: Linux guest-to-SVSM bridge

The source trace establishes a layered adapter path. `tpm2-tools` talks to `/dev/tpmrm0`; the Linux TPM subsystem invokes `tpm_svsm_send()`; the driver constructs a contiguous SVSM

vTPM request; the architecture-specific helper selects protocol 2 and supplies the buffer physical address; and the generic call layer chooses GHCB or MSR transport.

At the deepest guest boundary, Linux sets `call_pending`, places arguments in fixed registers, and executes `rep; vmmcall`. The host arranges VMPL0 execution, Coconut-SVSM dispatches the service, and Linux checks the pending flag and returned registers.

7.2 Cost versus security value

7.2.1 What changes in the trust model

Moving a vTPM from the host into VMPL0 has a concrete security purpose. A conventional host-emulated vTPM may be isolated from the guest but remains exposed to the host operator. An SVSM vTPM is placed inside the CVM's encrypted context and can keep its pages inaccessible to the lower-privilege guest. This aims to protect vTPM state from both sides of the interface.

The architecture redistributes trust rather than eliminating it. The AMD CPU and secure firmware remain trusted, while Coconut-SVSM becomes a highly privileged part of the guest trusted computing base. Its vTPM implementation and communication logic must be correct. The host remains trusted for availability but not for the confidentiality or integrity of private state.

This redistribution can be valuable even if the trusted code grows. The host kernel, QEMU, device emulators, and provider operations form a very large and externally controlled stack. Replacing trust in that stack with trust in a smaller, measured in-CVM module gives the owner a more direct object to inspect and attest. The smaller module still remains complex and security-critical. VeriSMo demonstrates why formal reasoning about VMPL0 software is useful even for Rust implementations [18].

7.2.2 Does the measured cost justify the service?

For the tested vTPM commands, the answer is favorable but scoped. A protocol region below 150 µs per command is small when the complete command takes approximately 30 ms. An extra few hundred kilobytes in Linux memory accounting is insignificant for an 8 GiB guest. These steady-state observations do not present an obvious deployment barrier.

The startup result is more consequential. The measured 5.988 s median difference could matter for short-lived workloads and rapid autoscaling. Even then, the decision depends on service lifetime. A multi-second one-time cost is severe for a function that runs for one second but minor for a VM that handles confidential data for a week. Furthermore, the large measured-boot debug dump means the observed number is likely sensitive to firmware build and serial configuration.

The practical trade-off is therefore workload-dependent, as shown in table 7.1.

7.2.3 Empirical advantages and disadvantages of the evaluated vTPM

Table 7.2 brings the architectural and empirical results together. The main advantage is not simply that the TPM interface works, but that its security-sensitive implementation and state move out of the provider-controlled host while remaining accessible through familiar Linux interfaces. The corresponding costs appear in the additional trusted code and integration needed to provide that service inside the CVM.

For the evaluated workload, the steady-state latency and guest-visible memory effects are secondary to startup behavior and system integration. The most immediate engineering priorities are earlier vTPM availability, protected persistence, and a clear binding between TPM evidence and the SNP-attested VM. This pattern is important because it shows where improving the architecture is likely to have more practical value than optimizing a transition that already occupies a small share of the tested command time.

Table 7.1: Expected relevance of the measured costs by workload type.

Workload	Most relevant cost	Interpretation
Long-lived confidential service	Runtime calls and memory	Measured steady-state costs are small for the selected TPM operations.
Autoscaled service	Boot and provisioning	Pre-kernel startup may affect scale-out response and should be measured under production firmware.
Serverless or short batch job	Boot amortization	Startup can dominate useful execution time. Snapshotting or pooling may be necessary.
TPM-intensive application	Call frequency and concurrency	The tested commands are low-frequency administrative operations; high-rate calls need separate load testing.
Disk unlock at boot	Boot plus persistence	A small number of TPM calls may be acceptable, but protected persistent state becomes essential.
Remote-attested service	Attestation path	Local PCR behavior is insufficient; evidence binding and verifier latency must be tested.

Table 7.2: Advantages and disadvantages of the evaluated in-CVM vTPM.

Dimension	Advantage	Disadvantage or remaining issue
Trust boundary	vTPM code and private state are removed from the untrusted host and isolated from the VMPL2 guest.	Coconut-SVSM and its TPM implementation become privileged parts of the CVM trusted computing base.
Compatibility	Linux exposed standard TPM devices, and existing `tpm2-tools` worked without a new userspace API.	The device registered too late for IMA in the evaluated boot, causing IMA to enter TPM-bypass.
Functionality	PCR operations, object creation, sealing, matching-policy release, and changed-PCR rejection worked.	Persistent state, composed remote attestation, broader command coverage, and formal conformance remain to be evaluated.
Runtime cost	Aggregate traced SVSM time was below about 0.5% of the tested userspace command time.	High-frequency, concurrent, and tail-latency behavior was outside the workload tested here.
Memory	The Linux-visible capacity difference was below 0.004% of an 8 GiB guest.	Complete VMPL0-private allocation requires a different measurement method.
Startup and deployment	The complete stack booted and delivered the service on physical SEV-SNP hardware.	The debug configuration added 5.988 s over SNP and required coordinated firmware, IGVM, QEMU/KVM, SVSM, and guest support.

7.3 What upstream integration says about readiness

The presence of an upstream Linux `tpm_svsm` driver is meaningful. It shows that the guest interface has moved beyond a private demonstration and can be represented through the kernel's standard TPM framework. Linux 6.16 integrated the enlightened vTPM support, while this thesis evaluated a Linux 7.0 guest [14, 19].

Upstream guest support is only one layer. The evaluated host kernel and QEMU still came from Coconut-SVSM branches, and the launch relied on a specific IGVM library and EDK2 branch. The project development plan lists continuing work on service isolation, persistence, device support, observability, and multiple execution modes [9]. These facts indicate active architectural progress, not a finished production stack.

The correct maturity judgment is therefore two-part:

- **Interface maturity is promising.** Standard Linux TPM devices and tools work, and the driver is upstream.
- **Deployment maturity is still evolving.** Host integration, firmware behavior, persistence, full attestation flows, and operational tooling require further validation.

7.4 Future work

7.4.1 Immediate measurement extensions

The most direct extensions follow from the limitations:

1. Repeat the 30-launch experiment with restored disk snapshots, randomized order, and a metadata record for host load.
2. Compare debug and release firmware, suppress the TPM event-log dump, and ablate vTPM or measured-boot activity to partition the long pre-kernel interval.
3. Preserve raw ftrace samples and report minimum, quartiles, p95, and maximum.
4. Compare the same TPM commands with a host vTPM or physical TPM to estimate relative overhead.
5. Trace KVM and Coconut-SVSM simultaneously to partition guest preparation, host dispatch, VMPL0 service, and return time.

7.4.2 Persistence and state continuity

Persistent vTPM state is the highest-value functional extension. It should test:

- whether keys, NV indices, and counters survive restart;
- how state is encrypted and authenticated outside the CVM;
- whether the host can replay an old valid state;
- whether one state file can be cloned into two CVMs;
- how rollback protection interacts with migration and disaster recovery.

7.4.3 Attestation composition

A remote verifier should request fresh evidence, validate the AMD certificate chain and SNP report, identify the Coconut-SVSM and vTPM service, and verify TPM evidence and measurement logs. The experiment should determine whether the verifier can bind all evidence to one current instance and reject replay.

7.4.4 Concurrency and reliability

Production workloads require concurrent commands, multiple vCPUs, cancellation, timeouts, and error recovery. Stress tests should measure throughput and tail latency while attempting malformed commands and forced service failures. Live migration adds further state and identity questions.

7.4.5 Other SVSM services

Linux also contains support paths for SVSM-assisted page validation, attestation, and vCPU lifecycle operations. Comparing their call counts and latency would show whether the low-frequency vTPM result generalizes to more frequent core services. The same call-path method used in this thesis provides a starting point.

7.5 Overall assessment

The evaluated stack demonstrates that a protected in-CVM vTPM can be reached through an ordinary Linux device interface and can enforce a meaningful PCR-bound policy end to end. Its strongest practical benefits are the improved placement of security-sensitive state and compatibility with existing TPM software. For the selected commands, the observed protocol latency and guest-visible memory effects were small.

The main disadvantages arose elsewhere: a larger trusted computing base, coordinated changes across several system layers, debug-instrumented startup cost, late vTPM registration relative to IMA, and unfinished persistence and attestation integration. Coconut-SVSM is therefore a functioning and rapidly developing implementation whose vTPM already demonstrates the value of privilege-layered CVMs, while also identifying the integration work needed to make that value dependable across the complete VM lifecycle.

Chapter 8

Related Work

8.1 Empirical studies of confidential virtual machines

Misono et al. provide a broad empirical comparison of AMD SEV-SNP and Intel TDX, covering computation, memory management, I/O, attestation, boot behavior, and trusted computing base considerations [16]. That study establishes that CVM cost depends on microarchitectural and software-stack details rather than one universal encryption penalty.

This thesis has a narrower target and a different level of focus. It studies one open-source SVSM stack inside SEV-SNP, follows a particular guest-to-service call path, and connects that path to vTPM behavior. The result complements broad CVM benchmarking by examining the additional privilege layer and service integration that exist after the CVM itself is available.

8.2 VMPL-based protected service frameworks

Veil uses VMPLs to place a security monitor inside a CVM and support protected services such as code integrity, logging, and shielded computation [20]. It addresses the limited number of VMPLs by combining them with x86 rings and replicating vCPU state across domains. Veil is a research framework designed to demonstrate flexible service isolation.

00SEVen uses a privileged in-VM agent to restore virtual-machine introspection for CVMs without trusting the provider or a compromised guest kernel [21]. It extends QEMU/KVM with VMPL-aware I/O and introspection assistance. This illustrates that the privileged layer can provide services that were traditionally rooted in the hypervisor, not only device emulation.

NestedSGX builds an enclave-like execution environment inside an SEV-SNP CVM using VMPLs and aims for compatibility with SGX toolchains [22]. Its reported context-switch costs also demonstrate why transition mechanisms need to be measured rather than assumed free.

These systems share the central architectural direction studied here: the CVM is internally partitioned so that selected code remains protected from both host software and the main guest operating system. Coconut-SVSM differs as a community-governed implementation of the AMD SVSM communication model with integration work across Linux, QEMU, and EDK2. This thesis evaluates that implementation rather than proposing a new isolation framework.

8.3 Verified security modules

VeriSMo is a verified security module for SEV-SNP that runs at VMPL0 and provides code integrity, runtime measurement, and secret-management services [18]. Its verification accounts for an untrusted hypervisor that can interrupt execution and modify permitted hardware state. The work shows that memory-safe implementation languages alone do not prove the correctness of a privileged module.

The present thesis uses empirical and architectural methods rather than formal verification. VeriSMo is directly relevant to the resulting discussion of trusted computing base quality because the SVSM holds more privilege than the guest kernel, so implementation defects can have broad consequences.

8.4 Confidential vTPMs

CoCoTPM proposes a TPM architecture for confidential VMs in which the host and hypervisor are excluded from the trusted computing base [23]. It emphasizes compatibility with existing TPM mechanisms for measured boot, disk encryption, TLS, and remote attestation.

Narayanan et al. implement an ephemeral vTPM inside SEV-SNP using VMPL isolation [17]. Their design avoids persistent host state and links the vTPM to the hardware root of trust for remote attestation. The work directly motivates the Coconut-SVSM vTPM direction and highlights state-management trade-offs.

Where these studies introduce vTPM architectures and attestation mechanisms, the present evaluation examines the existing Coconut-SVSM and upstream Linux interface on a concrete machine. It asks what the implementation can do, what its guest-observed calls cost, and whether a PCR-bound local policy works through standard tools.

8.5 Standards and open-source integration

The AMD SVSM specification defines discovery, Calling Area semantics, register conventions, core services, attestation, and the vTPM protocol [4]. The GHCB specification defines the guest-hypervisor communication format used around VMGEXIT [8]. The TPM 2.0 Library Specification defines the command and object model consumed by the vTPM [1].

Coconut-SVSM connects these standards to an open-source stack [5]. Linux's `tpm_svsm` driver exposes the service through the standard TPM class [14]. The Confidential Computing Consortium's hosting of Coconut-SVSM and the participation of several industry organizations indicate that the project is intended as shared infrastructure rather than a private prototype [6].

Standards and community adoption show ecosystem development, while performance and security behavior still depend on direct evidence. This thesis therefore treats ecosystem progress as context and uses experiments for the claims about the evaluated configuration.

8.6 Future Coconut-SVSM use cases and expected cost pressures

Coconut-SVSM's development plan describes a platform broader than the vTPM evaluated in this thesis. It includes user-mode service isolation, a UEFI variable-store service, persistent protected storage, paravisor support for less-enlightened operating systems, device abstractions, multi-platform support, and a service-VM mode in which one protected environment serves other CVMs [9]. These directions share the same basic trust goal, but they place pressure on different parts of the system.

Table 8.1 connects those planned directions to the cost most likely to matter in each case. The cost priorities are derived from the service path rather than reported as measurements of features that are not part of this evaluation.

In the near term, the vTPM is Coconut-SVSM's most concrete use case. It is implemented, has an upstream Linux interface, preserves a familiar API, and addresses a direct conflict between conventional host device emulation and the CVM trust model. UEFI variable storage and other stateful virtual devices follow the same security logic, but make persistence and rollback protection even more central. Full paravisor and service-VM modes could broaden Coconut-SVSM's role substantially; as request volume and device responsibility grow, throughput, concurrency, and

tail latency are likely to become more important than they were in the low-frequency vTPM workflow evaluated here.

Table 8.1: Likely cost and integration priorities for future Coconut-SVSM use cases.

Use case or direction	Security purpose	Likely dominant concern
vTPM and secret sealing	Protect keys, measurements, and policy state from the host and guest kernel.	Persistent state, rollback protection, early registration, and attestation binding.
UEFI variable-store service	Keep security-sensitive firmware variables inside the CVM trust boundary.	Persistence, rollback resistance, and added work on the boot path.
Short-lived and autoscaled CVMs	Provide protected services to workloads whose useful lifetime is brief.	Cold-start latency and the ability to amortize or avoid repeated initialization.
Full paravisor mode	Handle CVM-specific operations for operating systems with limited confidential-computing support.	Frequent transitions, device-emulation throughput, tail latency, and batching of fine-grained requests.
Protected persistent storage	Preserve service and virtual-device state outside volatile CVM memory.	Encryption and integrity cost, I/O throughput, recovery, and rollback prevention.
Attestation services	Bind hardware evidence to the SVSM, guest, and service state presented to a verifier.	Evidence-generation latency, freshness, verifier processing, and unambiguous evidence composition.
Service-VM mode	Provide protected services from one SVSM environment to other CVMs.	Secure cross-VM transport, concurrency, isolation between clients, throughput, and availability.

8.7 Position of this thesis

Prior work largely falls into three groups: broad characterization of CVM hardware, proposals for protected in-CVM services, and secure vTPM designs. This thesis sits between these groups. It studies an open-source system that implements the same architectural direction, but focuses on the engineering boundary from an ordinary Linux TPM command to a VMPL0 service and on the observable costs of using that system. The future-use-case analysis then applies these observations to the project's planned evolution, identifying when startup, per-call latency, persistence, storage throughput, or concurrency is most likely to become the limiting concern.

The work is also intentionally diagnostic. It distinguishes a deployment-level boot comparison from an isolated SVSM cost, a compound protocol duration from a raw VMPL switch, and Linux memory accounting from complete paravisor memory. These distinctions make the empirical findings reusable even when the code base changes.

Chapter 9

Conclusion

A TPM is a foundational component of modern platform security because it protects keys, records measured state, seals secrets to policy, and supports attestation. Virtual machines require equivalent functionality through a vTPM, but the conventional choice of placing that vTPM in the host conflicts with a confidential VM that treats the host as untrusted. Coconut-SVSM addresses this conflict by placing the vTPM at VMPL0 inside the protected SEV-SNP guest context, while isolating it from the ordinary guest operating system.

This thesis made that design concrete through an architectural and empirical characterization on physical AMD SEV-SNP hardware. It assembled and documented the complete open-source stack, traced a command from `tpm2-tools` through the Linux TPM subsystem and `tpm_svsm` driver to the Calling Area and VMGEXIT boundary, measured operational costs at startup and runtime, examined guest-visible memory accounting, and evaluated a representative PCR-bound sealing workflow.

Across three representative TPM commands, median observed call latency ranged from 35.9 µs to 49.1 µs. The aggregate traced time per command ranged from approximately 112 µs to 147 µs, less than approximately 0.5% of the observed 30 ms userspace command time. Linux-visible memory accounting changed by only 28 KiB in total memory and approximately 252 KiB in reserved memory, both below 0.004% of the configured guest size.

The vTPM evaluation demonstrated more than command connectivity. Standard tools modified PCR state, created and loaded sealed objects, recovered an unprotected secret, released a PCR-bound secret when the policy matched, and rejected the old policy after the PCR changed. Both PCR extensions exactly matched independent hash-chain calculations. The result shows that the post-registration interface is already useful for a representative local measured-state secret-release workflow. The IMA TPM-bypass observed before device registration also identifies earlier vTPM availability as an important next integration step.

Repeated boot measurements recorded medians of 9.386 s, 14.516 s, and 20.504 s for normal, SNP, and SNP with SVSM configurations. Coconut-SVSM added 5.988 s over the SNP baseline. Approximately four-fifths of this difference appeared between BDS start and Linux entry, an interval containing a median of 2,106 timestamped lines in SVSM versus 18 in SNP. The measured launch cost therefore characterizes the complete debug configuration, including its measured-boot logging, and is not an isolated cost of SVSM execution.

These findings expose a clear trade-off. The in-CVM vTPM improves the trust model and preserves compatibility with standard Linux software, while adding trusted VMPL0 code and requiring coordinated support across firmware, IGVM, QEMU/KVM, Coconut-SVSM, and the guest kernel. For the tested workload, steady-state call latency and guest-visible memory were not the main obstacles. Debug-build startup, early registration, persistence, remote attestation composition, and concurrency are more immediate engineering concerns.

The vTPM is likely to remain Coconut-SVSM's most concrete near-term use case because it is implemented, upstream-facing, and solves a specific confidential-computing trust problem.

Planned UEFI variable storage and other stateful services extend the same idea, while paravisor and service-VM modes could make Coconut-SVSM responsible for a much larger share of CVM operation. In those broader roles, cold-start cost, transition frequency, device throughput, tail latency, secure storage, and concurrency will require greater attention.

Overall, Coconut-SVSM demonstrates a practical direction for confidential computing: a CVM can contain its own hardware-isolated service layer instead of returning security-sensitive services to the untrusted host. The evaluated vTPM shows that this direction can provide meaningful security functionality through familiar software interfaces, and the measurements identify where the implementation can be improved as its role expands.

Bibliography

[1] Trusted Computing Group, "TPM 2.0 Library Specification." https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2026. Accessed July 24, 2026.

[2] Advanced Micro Devices, Inc., "AMD Secure Encrypted Virtualization (SEV)." https://www.amd.com/en/developer/sev.html, 2026. Accessed July 24, 2026.

[3] Advanced Micro Devices, Inc., "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," Tech. Rep. 70366, Advanced Micro Devices, Inc., Jan. 2020.

[4] Advanced Micro Devices, Inc., "Secure VM Service Module for SEV-SNP Guests: Guest Communication Interface," Tech. Rep. 58019, Advanced Micro Devices, Inc., Mar. 2026.

[5] COCONUT-SVSM Project, "COCONUT Secure VM Service Module." https://github.com/coconut-svsm/svsm, 2026. Commit 44f639d2efbe481c5ba15a152571da90e5271cb0 evaluated in this thesis.

[6] Confidential Computing Consortium, "COCONUT-SVSM Joins the Confidential Computing Consortium." https://confidentialcomputing.io/2024/07/02/coconut-svsm-joins-theconfidential-computing-consortiumenhancing-security-for-sensitiveworkloads/, July 2024. Accessed July 24, 2026.

[7] Advanced Micro Devices, Inc., "SEV Secure Nested Paging Firmware ABI Specification," Tech. Rep. 56860, Advanced Micro Devices, Inc., May 2025.

[8] Advanced Micro Devices, Inc., "SEV-ES Guest-Hypervisor Communication Block Standardization," Tech. Rep. 56421, Advanced Micro Devices, Inc., Jan. 2025.

[9] COCONUT-SVSM Project, "Development Plan Overview." https://coconut-svsm.github.io/svsm/developer/DEVELOPMENT-PLAN/, 2026. Accessed July 24, 2026.

[10] Microsoft, "Independent Guest Virtual Machine (IGVM) Format." https://github.com/microsoft/igvm, 2026. Commit fc0fa3f55cde54e016a1169926ae39f9561aa8f7 evaluated in this thesis.

[11] COCONUT-SVSM Project, "Installing the COCONUT-SVSM." https://coconut-svsm.github.io/svsm/installation/INSTALL/, 2026. Accessed July 24, 2026.

[12] tpm2-software Community, "tpm2-tools." https://github.com/tpm2-software/tpm2-tools, 2026. Accessed July 24, 2026.

[13] tpm2-software Community, "tpm2_policypcr Manual Page." https://tpm2-tools.readthedocs.io/en/latest/man/tpm2_policypcr.1/, 2026. Accessed July 24, 2026.

[14] Linux Kernel Developers, "SNP SVSM vTPM Driver: drivers/char/tpm/tpm_svsm.c." https://codebrowser.dev/linux/linux/drivers/char/tpm/tpm_svsm.c.html, 2025. Upstream Linux source.

[15] Linux Kernel Developers, "ftrace: Function Tracer." https://docs.kernel.org/trace/ftrace.html, 2026. Accessed July 24, 2026.

[16] M. Misono, D. Stavrakakis, N. Santos, and P. Bhatotia, "Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 8, no. 3, 2024.

[17] V. Narayanan, C. Carvalho, A. Ruocco, G. Almási, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev, "Remote Attestation of Confidential VMs Using Ephemeral vTPMs," in *Annual Computer Security Applications Conference*, 2023.

[18] Z. Zhou, Anjali, W. Chen, S. Gong, C. Hawblitzel, and W. Cui, "VeriSMo: A Verified Security Module for Confidential VMs," in *18th USENIX Symposium on Operating Systems Design and Implementation*, pp. 599–614, USENIX Association, 2024.

[19] Linux Kernel Newbies, "Linux 6.16." https://kernelnewbies.org/Linux_6.16, 2025. Release summary listing enlightened vTPM support for SVSM on SEV-SNP.

[20] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca, "Veil: A Protected Services Framework for Confidential Virtual Machines," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 378–393, 2023.

[21] F. Schwarz and C. Rossow, "00SEVen: Re-enabling Virtual Machine Forensics: Introspecting Confidential VMs Using Privileged in-VM Agents," in *33rd USENIX Security Symposium*, pp. 1651–1668, USENIX Association, 2024.

[22] W. Wang, L. Song, B. Mei, S. Liu, S. Zhao, S. Yan, X. Wang, D. Meng, and R. Hou, "The Road to Trust: Building Enclaves within Confidential VMs," in *Network and Distributed System Security Symposium*, 2025.

[23] J. Pecholt and S. Wessel, "CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments," in *Annual Computer Security Applications Conference Workshops*, 2022.

Appendix A

Reproducibility Details

This appendix consolidates the implementation identifiers, configuration differences, and data-collection rules used in the evaluation. It is intended to make the experiments repeatable without forcing the main methodology chapter to carry every command-line detail.

A.1 Artifact identifiers

Table A.1 lists the software revisions used in the evaluated stack. The identifiers should be retained even if a later rerun changes only the measurement script. If an implementation component is rebuilt from a different revision, the new identifier should be recorded as a separate experiment rather than silently replacing the original.

Table A.1: Software and artifact identifiers.

Component	Version or size	Revision or digest
Host Linux	6.11.0+	`3d7f4e43fb9b1b41f1f0ef09c8f6d770505e59ce`
QEMU	development build	`34d36846c6bc670bdb632b94be11d386d4c925d0`
Coconut-SVSM	2026.04 development series	`44f639d2efbe481c5ba15a152571da90e5271cb0`
IGVM tools	development build	`fc0fa3f55cde54e016a1169926ae39f9561aa8f7`
EDK2/OVMF	development build	`5035a8789f556bd189a22c3aa594bb9048111461`
Guest Linux	7.0.0-22-generic	Ubuntu 26.04 LTS
Rust and Cargo	1.88.0	Build toolchain
LLVM	20.1.5	Build toolchain
GCC	13.3.0	Build toolchain
GNU binutils	2.42	Build toolchain
IGVM image	4,831,344 bytes	SHA-256: `d65d90c850f424f46241b33b47bc47798c2dc51e532ab8d96419bc14e0b8b9d0`

A.2 Launch-configuration differences

All configurations use KVM acceleration, QEMU's `q35` machine type, the `EPYC-v4` CPU model, eight vCPUs, 8 GiB of memory, the same guest disk, and the same cloud-init seed. Table A.2

isolates the parameters that intentionally differ.

Table A.2: Intentional differences among launch configurations.

Parameter	Normal VM	SNP	SNP with SVSM
Confidential guest object	None	`sev-snp-guest`	`sev-snp-guest`
Memory backend	Conventional QEMU RAM	Shared memfd backend	Shared memfd backend
Firmware delivery	OVMF through `-bios`	OVMF through `-bios`	IGVM through `igvm-cfg`
SVSM at VMPL0	No	No	Yes
Guest execution level	Conventional VM	VMPL0 guest	Linux at VMPL2
vTPM path	None in this test	None in this test	`tpm_svsm`

The firmware-delivery difference is architecturally necessary for this setup, but it also means that the boot comparison is a full-stack deployment comparison. It is not a controlled subtraction of only SVSM computation.

A.3 Representative QEMU invocations

The following listings reproduce the critical launch options while replacing machine-specific home-directory prefixes with descriptive environment variables. The paths should resolve to the exact revisions listed in Table A.1.

Listing A.1: Common variables for the launch commands.

```
QEMU_BIN="$HOME/bin/qemu-svsm/bin/qemu-system-x86_64"
OVMF="$HOME/thesis/edk2/Build/OvmfX64/DEBUG_GCC/FV/OVMF.fd"
IGVM="$HOME/thesis/svsm/bin/coconut-qemu.igvm"
IMAGE_DISK="$HOME/thesis/image/svsm-golden-setup.qcow2"
SEED="$HOME/thesis/seed/seed.iso"
IGVM_LIB="$HOME/igvminst/lib/x86_64-linux-gnu"
```

Listing A.2: Core SNP launch options.

```
sudo env LD_LIBRARY_PATH="$IGVM_LIB" "$QEMU_BIN" \
  -enable-kvm -cpu EPYC-v4 \
  -machine q35,confidential-guest-support=sev0,memory-backend=ram1 \
  -object memory-backend-memfd,id=ram1,size=8G,share=true,prealloc=false,reserve=
    false \
  -object sev-snp-guest,id=sev0,cbitpos=51,reduced-phys-bits=1 \
  -bios "$OVMF" -smp 8 -no-reboot \
  -netdev user,id=vmnic,hostfwd=tcp::2223-:22 \
  -device e1000,netdev=vmnic,romfile= \
  -drive file="$IMAGE_DISK",if=none,id=disk0,format=qcow2,snapshot=off \
  -drive file="$SEED",format=raw,if=virtio,readonly=on \
  -device virtio-scsi-pci,id=scsi0,disable-legacy=on,iommu_platform=on \
  -device scsi-hd,drive=disk0,bootindex=0 \
  -vga none -serial stdio -serial pty
```

Listing A.3: Core SNP-with-SVSM launch options.

```
sudo env LD_LIBRARY_PATH="$IGVM_LIB" "$QEMU_BIN" \
  -enable-kvm -cpu EPYC-v4 \
  -machine q35,confidential-guest-support=sev0,memory-backend=ram1,igvm-cfg=igvm0 \
  -object memory-backend-memfd,id=ram1,size=8G,share=true,prealloc=false,reserve=
    false \
  -object sev-snp-guest,id=sev0,cbitpos=51,reduced-phys-bits=1 \
  -object igvm-cfg,id=igvm0,file="$IGVM" \
  -smp 8 -no-reboot \
  -netdev user,id=vmnic,hostfwd=tcp::2222-:22 \
  -device e1000,netdev=vmnic,romfile= \
  -drive file="$IMAGE_DISK",if=none,id=disk0,format=qcow2,snapshot=off \
  -drive file="$SEED",format=raw,if=virtio,readonly=on \
  -device virtio-scsi-pci,id=scsi0,disable-legacy=on,iommu_platform=on \
  -device scsi-hd,drive=disk0,bootindex=0 \
  -vga none -serial stdio -serial pty
```

The normal-VM command removes the confidential-guest object and its machine reference, uses conventional QEMU memory, and retains OVMF through `-bios`. Apart from those required differences, its CPU, vCPU count, guest memory, disk, network device, and serial settings are aligned with the two protected configurations.

A.4 Repeated boot protocol

The final boot dataset contains ten successful observations for each of the three configurations. The runs were collected in configuration blocks in the order SVSM, SNP, then normal VM. This is reported because a block design can confound configuration with temporal drift. The launch scripts used the same guest image and `snapshot=off`.

For every observation, the procedure is:

1. confirm that no process from the previous QEMU instance remains;
2. launch the selected configuration with the fixed command and disk policy;
3. record relative timestamps with one persistent reader;
4. record the exact `CVM_READY` marker;
5. shut down the guest cleanly and confirm that QEMU exits;
6. preserve the complete timestamped serial log.

The primary statistic is the median time to `CVM ready`. Variability is reported using the interquartile range. All individual observations are retained. The 63.803 s SNP run is included and shown in Figure 5.1; its delay is localized in the raw log rather than removed as an inconvenience.

The analysis script in `analysis/parse_coldboot.py` validates that each log contains the BDS, Linux, systemd, and ready markers in order. It emits:

- `coldboot_runs.csv`, containing every endpoint, phase duration, and BDS-to-Linux line count;
- `coldboot_summary.csv`, containing the endpoint descriptive statistics; and
- `coldboot_phase_summary.csv`, containing per-phase descriptive statistics.

A.5 Trace and memory records

For the latency experiment, the raw ftrace stream must retain the entry and return timestamps for `svsm_perform_ghcb_protocol()`. The analysis groups calls by userspace invocation before computing both per-call latency and aggregate traced time per command. The 50 invocations of each selected command should remain available with the derived table so that medians can be recalculated.

For memory accounting, the complete Linux `Memory:` line is retained for every observation. The comparison uses Linux-reported total memory and the mean reserved-memory value. It must not be relabeled as the complete SVSM allocation, because VMPL0-private pages may not be represented as an ordinary Linux reservation.

A.6 Central result definitions

Final boot statistics and exact vTPM values are defined once in `thesis_data.tex`. The abstract, result tables, discussion, and conclusion reuse those commands. This prevents small numerical inconsistencies during revision.

Appendix B

vTPM Functional Test Commands

This appendix records the exact command sequence used for the representative vTPM workflow. The main evaluation chapter summarizes the test intentions and outcomes. The appendix provides the operational detail needed to reproduce them.

B.1 Setup and device visibility

Listing B.1: Workspace setup and basic vTPM discovery.

```
cd ~/vtpm_verify/pcr_seal_test_20260724_093109

date --iso-8601=seconds
uname -a
tpm2_getcap --version
ls -l /dev/tpm*
lsmod | grep -iE "tpm|svsm" || true
sudo dmesg | grep -iE "tpm|svsm" || true
sudo tpm2_getcap properties-fixed
sudo tpm2_getrandom 16 | xxd
sudo tpm2_pcrread sha256:0,1,2,3,4,5,7
```

This step checks that the guest exposes the TPM character devices, that the kernel reports the SVSM-backed path, and that standard `tpm2-tools` commands can query properties, obtain random bytes, and read PCRs.

B.2 PCR state transition

Listing B.2: PCR 16 reset and extension.

```
sudo tpm2_pcrread sha256:16
sudo tpm2_pcrreset 16
sudo tpm2_pcrread sha256:16

echo -n "first measurement" > measurement1.txt
DIGEST1=$(sha256sum measurement1.txt | awk '{print $1}')
echo "$DIGEST1"

sudo tpm2_pcrextend 16:sha256=$DIGEST1
sudo tpm2_pcrread sha256:16
```

The captured reset value was:

```
0x0000000000000000000000000000000000000000000000000000000000000000
```

The captured post-extension value was:

```
0xDBDFD380040992F64BBB35B17D3BE7B74A496B8B13D898844A9ED4906390C949
```

The latter exactly matched an independent calculation of the PCR extend formula.

B.3 Basic sealing and unsealing

Listing B.3: Unrestricted seal and unseal test.

```
sudo tpm2_createprimary -C o -G ecc -g sha256 -c primary.ctx
echo -n "svsm-vtpm-secret" > secret.txt

sudo tpm2_create \
  -C primary.ctx \
  -u seal.pub \
  -r seal.priv \
  -i secret.txt

sudo tpm2_load \
  -C primary.ctx \
  -u seal.pub \
  -r seal.priv \
  -c seal.ctx

sudo tpm2_unseal -c seal.ctx -o unsealed.txt
sudo chown "$USER:$USER" unsealed.txt
cat unsealed.txt
cmp secret.txt unsealed.txt && echo "basic seal/unseal PASS"
```

The final comparison verifies the output byte-for-byte. A successful `tpm2_unseal` invocation without this comparison would establish only that a command returned successfully, not that it released the intended value.

B.4 PCR-bound policy creation

Listing B.4: Creation of a policy tied to SHA-256 PCR 16.

```
sudo tpm2_pcrreset 16
sudo tpm2_pcrread sha256:16
sudo tpm2_pcrread -o pcr16.bin sha256:16

sudo tpm2_startauthsession -S trial_session.ctx
sudo tpm2_policypcr \
  -S trial_session.ctx \
  -l sha256:16 \
  -f pcr16.bin \
  -L pcr16.policy
sudo tpm2_flushcontext trial_session.ctx
sudo xxd -p -c 32 pcr16.policy
```

The binary PCR record and generated policy file are supporting artifacts. They do not need to be printed in full in the thesis, but retaining them connects the policy input to the later authorization test.

B.5 Matching-policy success case

Listing B.5: PCR-bound seal and matching-state unseal.

```
echo -n "pcr-bound-svsm-secret" > pcr_secret.txt

sudo tpm2_create \
  -C primary.ctx \
  -u pcr_seal.pub \
  -r pcr_seal.priv \
  -L pcr16.policy \
  -i pcr_secret.txt

sudo tpm2_load \
  -C primary.ctx \
  -u pcr_seal.pub \
  -r pcr_seal.priv \
  -c pcr_seal.ctx

sudo tpm2_startauthsession \
  --policy-session \
  -S policy_session.ctx
sudo tpm2_policypcr \
  -S policy_session.ctx \
  -l sha256:16 \
  -f pcr16.bin

sudo tpm2_unseal \
  -c pcr_seal.ctx \
  -p session:policy_session.ctx \
  -o pcr_unsealed.txt
sudo tpm2_flushcontext policy_session.ctx
sudo chown "$USER:$USER" pcr_unsealed.txt

cmp pcr_secret.txt pcr_unsealed.txt \
  && echo "PCR-bound unseal with matching PCR: PASS"
```

This is the positive authorization case. The policy session is satisfied while the live PCR value agrees with the PCR record used to construct the policy.

B.6 Changed-policy negative case

Listing B.6: PCR change followed by the expected policy rejection.

```
echo -n "changed measurement" > measurement2.txt
DIGEST2=$(sha256sum measurement2.txt | awk '{print $1}')
echo "$DIGEST2"

sudo tpm2_pcrextend 16:sha256=$DIGEST2
sudo tpm2_pcrread sha256:16
sudo tpm2_startauthsession \
  --policy-session \
  -S negative_session.ctx

if sudo tpm2_policypcr \
  -S negative_session.ctx \
  -l sha256:16 \
```

```
  -f pcr16.bin
then
  echo "ERROR: policy unexpectedly satisfied"
  NEGATIVE_RESULT=1
else
  echo "EXPECTED: policy rejected changed PCR"
  NEGATIVE_RESULT=0
fi

sudo tpm2_flushcontext negative_session.ctx || true
test "$NEGATIVE_RESULT" -eq 0 \
  && echo "PCR-BOUND NEGATIVE TEST: PASS"
```

In the observed run, rejection occurred while satisfying `tpm2_policypcr`, so execution did not proceed to secret release. This is the expected security outcome. The negative test is stronger than a sequence containing only successful commands because it checks that the authorization condition is enforced after the protected state changes.

B.7 Recorded evidence summary

The July 24 capture records Linux 7.0.0-22, `tpm2-tools` 5.7, the default `tcti-device` interface, both TPM character devices, and the `tpm_svsm` module. Its exact digest records are:

```
post-extension PCR:
0xDBDFD380040992F64BBB35B17D3BE7B74A496B8B13D898844A9ED4906390C949

policy digest:
bff2d58e9813f97cefc14f72ad8133bc7092d652b7c877959254af140c841f36

changed PCR:
0x43DB69D614F6862F5FF258B5D359345325725C1D5BCB91BA53271E0461C7CC09
```

The capture also preserves successful byte comparisons for basic and PCR-bound unseal, followed by the expected `tpm2_policypcr` error `0x1C4` and `PCR-BOUND NEGATIVE TEST: PASS`.

The full evidence document is retained with the thesis package rather than printed in full here. It also records the early IMA TPM-bypass message and explains the root-owned unseal output-file artifact.

Appendix C

Claim-to-Evidence Matrix

Table C.1 makes the evidential boundaries of the thesis explicit. It separates direct observations from interpretation and records wording that the experiments do not justify. This matrix is also a consistency checklist for later edits.

Table C.1: Claim-to-evidence matrix for the evaluated Coconut-SVSM stack.

Topic	Direct evidence	Defensible interpretation	Unsupported extension
Architecture	Source-level trace from the Linux TPM subsystem through `tpm_svsm`, the SVSM Calling Area, and `svsm_perform_ghcb_protocol()`.	The evaluated guest exposes the SVSM vTPM through the standard Linux TPM interface while using a VMPL-separated request path.	The complete path has been formally verified, or every Linux TPM workload follows an identical dynamic path.
VMPL transition cost	ftrace durations around `svsm_perform_ghcb_protocol()` for 50 invocations of each of three commands.	The guest-observed GHCB-based SVSM protocol region took tens of microseconds per call in this setup.	The number is the isolated hardware cost of changing VMPL, with no driver, protocol, scheduling, or SVSM work included.
Command-level significance	Median aggregate traced time of 112–147 microseconds against an approximate 30-millisecond userspace duration.	The traced region was not the dominant latency component for the three tested commands.	Coconut-SVSM adds less than 0.5 percent overhead to TPM workloads in general.
Memory capacity	Linux reported 8,382,640 KiB total for SNP and 8,382,612 KiB for SNP with SVSM; mean reserved values differed by about 252 KiB.	Coconut-SVSM had a negligible effect on Linux-visible memory capacity in the evaluated 8-GiB configuration.	The complete static or runtime footprint of Coconut-SVSM is exactly 28 KiB or 252 KiB.
Boot phase	Across ten runs per configuration, the median BDS-to-Linux interval was 0.047 s for SNP and 4.793 s for SVSM; median line counts were 18 and 2,106.	Most of the observed complete-stack SNP-to-SVSM difference is localized before Linux entry and coincides with high-volume measured-boot debug output.	The interval is the intrinsic cost of VMPL switching, vTPM cryptography, or a production release build.
Repeated boot result	Thirty raw logs produced medians of 9.386 s, 14.516 s, and 20.504 s for normal, SNP, and SVSM, with all observations retained.	The data characterizes complete-stack time to the fixed ready marker under the tested debug configuration.	The result isolates only SVSM computation from firmware packaging, logging, storage, and other configuration differences.

Topic	Direct evidence	Defensible interpretation	Unsupported extension
Device integration	`/dev/tpm0`, `/dev/tpmrm0`, the `tpm_svsm` driver, and successful standard `tpm2-tools` commands.	The evaluated vTPM integrates with the conventional Linux TPM userspace interface.	Every existing TPM application is compatible without modification.
PCR operations	PCR 16 reset, read, extension, and changed value.	The evaluated implementation supports the tested PCR state operations.	All PCR banks, indices, locality rules, and edge cases conform to the complete TPM specification.
Sealing	Primary-object creation, sealed-object creation and loading, successful unseal, and byte comparison.	The vTPM supports the representative sealing workflow used in the experiment.	All TPM object types, authorization modes, and persistence behaviors are complete.
Policy enforcement	PCR-bound release succeeded with matching state; after an independently verified PCR change, policy satisfaction failed with TPM error 0x1C4.	The implementation enforced the tested PCR-bound authorization condition after driver registration.	The design provides rollback protection, persistence across all lifecycle events, or complete remote-attestation security.
Functional maturity	A coherent workflow covering discovery, capabilities, randomness, PCR state, object management, sealing, and a negative policy case; IMA entered TPM-bypass before device registration.	The vTPM has meaningful post-registration functional coverage and supports a representative local secret-release use case.	Formal TPM 2.0 compliance, a production-ready drop-in replacement, or complete ecosystem compatibility.
Generalizability	One hardware platform and a documented set of software revisions and VM parameters.	The findings are an implementation-centered characterization of this stack.	Identical performance on other processors, hosts, kernels, hypervisors, or future Coconut-SVSM revisions.

C.1 Use during final revision

Every quantitative statement in the abstract and conclusion has a corresponding row in this matrix and a result in the main evaluation chapters. The broader unsupported statements remain unsupported unless a new experiment directly addresses them.

This discipline does not weaken the thesis. It makes the main contribution clearer: a technically detailed characterization of one emerging open-source confidential-computing stack, supported by measurements at several system levels and a functional security workflow.