DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluating the Performance Impact of Privilege Levels of Confidential Virtual Machines

Alexander Schindler

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Evaluating the Performance Impact of Privilege Levels of Confidential Virtual Machines

Evaluierung der Leistungsauswirkungen von Privilegien-Stufen auf vertrauliche virtuelle Maschinen

Author:	Alexander Schindler
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Patrick Sabanic
Submission Date:	15.11.2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.11.2024 Alexander Schindler

Acknowledgments

I want to thank my girlfriend Becky for always supporting me through this stressful time. I also want to thank Patrick for his patience with me. Lastly I need to also give credit to Teto, my emotional support hamster.

Abstract

This thesis gives insights into a hardware platform by AMD that enables VMs to execute in a confidential manner, even when the host platform is in possession and control of a third party. This platform is called SEV-SNP. It allows the guest to encrypt all it's memory usage as well as protect against integrity. Additionally, the concept of VM privilege levels is introduced. It allows encryption in levels; i.e. a less privileged program can be prevented any access to memory pages of a more privileged one, enforced by hardware mechanisms. This also includes access from the hypervisor, meaning the outside world. A security monitor has been specified on top of that technology, allowing even an OS kernel (like Linux) to be secured from the outside. The monitor runs at a higher VM privilege level than the guest OS. An extension is proposed that can be useful in hardening the VM from an attack that swaps the guest kernel image upon boot. Furthermore benchmarks are conducted to measure the performance of VMs that incorporate SVSM. They are then being compared with VMs that do not have SVSM active.

Contents

1. Introduction

For many modern businesses self-hosting all their infrastructure has become less and less attractive. Sometimes it is not possible to predict the demand for their services and resources. And since economic growth as well as a growth in demand is a direct goal, more flexibility in their hardware configuration is desirable. This flexibility cannot be offered by a fixed hardware installation that is purchased as a commodity. Therefore, the main question for those classical setups focuses on which hardware should be purchased and determining the adequate amount. Capacity (computing power, memory, ...) will either be too large, with future growth in mind, or too small due to budget constraints.

This could generate a great waste of financial resources, in various situations. What if the growth is slower than anticipated and/or the hardware is too outdated once higher demand needs to be met? On the other hand, what if capacity insufficient too quickly? Such situation could cause bottlenecks and denial of service to some or even all users if growth in demand for services exceed expectations. Even if the hardware's intended use would be restricted to internal purposes (e.g., development) it might not be trivial to determine which and how much hardware should be purchased.

Due to significant costs for such hardware a tradeoff needs to be made, trying to account for uncertainty and the development of better hardware in the foreseeable future. On top of that, the popularity of the internet allows for quick shifts in demand. A web-shop, for example, may suddenly gain popularity due to different factors, such as social media trends, which would derive into a short burst in demand when discounts are offered or during a holiday. A modern and natural solution to this problem is to switch to cloud hosted systems. A variety of services and deployment models have been released in the past years. However, they all offer a pay-per-use model with high flexibility and scalability. Furthermore that hardware will be upgradeable more frequently and maintenance is being outsourced. This ideally accounts for the needs of most businesses for their infrastructure and deployment models. Decisions for purchasing cloud capacities are easier to make than decisions for acquiring self-hosted hardware, which involves a lot of on-site expertise, planning with uncertain factors and reasoning. Thus, it is not surprising that cloud providers have proven to build on a successful business model.

However, there is a big question about data security when it comes to outsourcing computing resources. Companies will deal with confidential data; whether that is sensitive user data, e.g. payment information and addresses, or internal, valuable business data. A classical cloud provisioned service may also require outsourcing some of the responsibility for such data. How much trust can be put into authentication

mechanisms used by the cloud provider? How secure are the configurations of the host and/or hypervisor? How easy is it for the cloud provider themselves to act maliciously? Whenever possible, critical data hence needs to be protected even from the hosting cloud provider as well as potential attacks against said provider or software faults on the side of the provider. Huge data centers with multi tenant systems that accumulate lots of data pose a favorable target for all sorts of attacks.

State-of-the-art cryptography is of limited help from the perspective of a customer. The connection to the service and transfer of data can be protected (e.g. via TLS or ssh), and the data written to disk can be encrypted, and its integrity secured by the use of modern crypto libraries within the client software. However, in memory and CPU registers all data needs to be loaded in plain text (including e.g. the disk encryption key). Although the two mentioned protections offer some security against the easiest attacks, there is still a significant risk in processing data on third party systems. Hence, companies currently need to naively trust cloud providers to protect their systems to the highest standards and to not break confidentiality themselves. If this risk is unacceptable, this otherwise attractive outsourcing of hardware can not be considered.

Recent chips from both Intel [6] and AMD [12] offer solutions to this problem. The approach is to offer hardware enforced encryption for marked physical pages in memory and mechanisms to attest this encryption is being enforced with a clients' virtual machine. This thesis will focus on AMD's SEV technology with the SNP extension [16]. Most importantly, this extension additionally provides encryption for the CPU registers (via the SEV-ES extension) and mechanisms to detect tempering with encrypted pages like swapping out an encrypted page for an unencrypted one. On top of that it also introduces so-called Virtual Machine Privilege Levels (VMPL). This is a mechanism orthogonal but analogous to the privilege rings that are used to separate user mode from kernel mode on Linux. For VMPL, a context within a VM can be loaded in one of four hierarchical privilege levels. To fully utilize the power of SEV-SNP a small (compared to the size of an OS kernel) security monitor is loaded into the highest of these privilege levels while the guest OS (in our case it will be Linux) is loaded in a lower privilege level. This may seem slightly unintuitive as an OS kernel usually has full access over the hardware resources. However, this design follows the principle of least privilege and the OS kernel is not tasked with ensuring SEV mechanisms are enforced. This also means an OS cannot set its memory pages into an unencrypted state, further hardening the VM by limiting the attack surface.

Currently, there is no fully fledged, production ready security monitor of such kind. The CoconutSVSM [3] project aims to implement the SVSM specification and offer all necessary features to the guest OS. Already a fair number of features is available, e.g. the whole specified startup mechanism, but it is still under active development. This thesis will add a protocol to the SVSM module to allow measuring physical memory contents of the guest OS. The protocol allows the guest OS to request this measurement for a certain page in memory (or even a range of pages). The SVSM will read the contents of the page(s), hash them and request for the Secure Processor (i.e. the hardware interface

for SEV-SNP) to sign this hash, which will then be returned as part of a report to the guest. The signature can then be verified remotely, and the hash can be compared to the expected value. On top of that a kernel module will be introduced that allows communication with the SVSM monitor and can verify guest provided memory pages. By offering this functionality, an additional security layer is added; given SVSM and the SEV-SNP features protect the guest from the outside world sufficiently during runtime, secure deployment can now also be monitored and verified.

On top of that, benchmarks are being conducted, comparing the performance of SVSM enabled guests with plain VMs. The motivation for this is given and results are discussed.

At first, chapter 2 offers insight into the used technologies and prerequisites, namely SEV-SNP, SVSM and the Linux kernel. Next, chapter 3 explains some of the attack vectors this feature extensions protects against and discusses the security model. chapter 4 and chapter 5 will showcase and explain the implementation details for this extension. Following that will be an evaluation of performance impacts and feasibility in chapter 6. Lastly, a conclusion is drawn in chapter 7.

2. Technology and Prerequisites

This section will explore the technology that ultimately enables the kernel verification. Some detailed knowledge is necessary to understand the mechanism that will be described in chapter 5. Namely, the three platforms shown here are AMD SEV-SNP, CoconutSVSM as well as a brief overview of the Linux kernel, especially regarding memory management. AMD SEV-SNP offers the underlying features for encryption in hardware. Only a small portion of this interface will be used, however understanding some basic ideas of SEV-SNP is crucial for context. It will be assumed that SEV-SNP itself is as secure as the specification states, however there will be some discussion about that in chapter 3. CoconutSVSM [3] is a work-in-progress implementation of AMD's SVSM specification [13]. It comprises a relatively small kernel implementing protocols that the guest OS can use to communicate with the higher privilege layer to request certain services. It also comes with early stage bootloaders that set up SVSM, verifies that SEV-SNP is active and encryption is enabled for the guest. The Linux kernel is well known and very large; its dedicated subsection will only focus on a few things that are being utilized in chapter 5.

2.1. Secure Encrypted Virtualization with Secure Nested Paging

AMD specified an interface for the context of confidential computing in three iterations that are relevant for this thesis. Confidential computing refers to the ability to run machines in the cloud under full confidentiality. As will be shown, such a computing model is simply not possible without hardware support.

Tenants on a cloud infrastructure could incorporate algorithms to encrypt all write-outs to storage mediums and use secured connections to transfer data and instructions to the compute node. This of course would offer some rudimentary security. However, everything within the bounds of the CPU, RAM and Data Buses is not secured in this way. As a customer of cloud compute resources this would mean a lot of components need to be trusted blindly. This includes the underlying hardware, the hypervisor and the security mechanisms that the cloud provider would enforce. If one of these components were to act maliciously, were exploited by a hostile entity (e.g. a malicious tenant on the same physical hardware) or simply had a bug or be misconfigured, private data from within the virtual machine could be leaked or even manipulated. A customer of the cloud service has no power over the framework onto which their machine is being deployed. There is no direct way of knowing if data was compromised or tempered with. Certainly, without confidential computing there is no direct way to prevent it.

The main purpose of confidential computing in the cloud is reducing the components that need to be trusted to exactly one: the CPU. This already mitigates a lot of attack vectors and potential sources of faults as described above. On top of that the trust into the CPU can be established via protocol. More details are to follow.

2.1.1. Iteration One: SEV

The first iteration was called SEV [8] and introduced the core principles for encrypting memory in virtual machines. It integrates with AMD's virtualization architecture (AMD-V) and transfers a concept AMD calls Secure Memory Encryption (SME). SME simply encrypts pages in memory in a performant way (since the algorithm and mechanism are embedded into the hardware). A page can be marked for encryption within the page tables and the memory controller will then enforce such encryption (and decryption when the page is accessed by the CPU). For the case of SEV, such encryption is conducted using a key that is assigned to a single VM. This key cannot be accessed by software (including the hypervisor) and SEV will make sure it only decrypts a memory page for usage if the corresponding VM is accessing it. While being decrypted within the CPU, SEV will keep track of what data belongs to which VM (or hypervisor for that matter) and uses hardware enforced access control. If multiple VMs live on the hardware node, each VM will be assigned its own key, as to encrypt each VM's memory independently of everything else on the physical system.

The guest OS needs a few modifications in order to utilize SEV features. The guest's page tables need to be managed to enable or disable encryption for certain pages. More precisely, the so called C-bit needs to be adjusted for physical addresses of the guest. SEV will then only encrypt those pages, that have their C-bit set. Secondly, the guest OS must disallow direct memory access for pages it wants to be encrypted. An attempt of DMA into an encrypted page will result in a fault by SEV.

As stated above, the crux for this platform to be useful is whether the CPU and the secure components within it can be trusted. Although the guest needs to be aware of the SEV features and marks its pages for encryption, it never sees the encryption happening. It is only the hardware that intercepts whenever pages are written to memory that could verify the security.

For that reason, another key feature of SEV is an attestation mechanism that can prove SEV is up and running correctly on a remote hardware node. The guest management API [12] provides a command to request a report, which returns a signed digest (together with some additional data). Ultimately, the guest only needs to trust the AMD root keys that set up the keychain for the signatures. Once the signature is checked and the contents of the report match the expected values, the guest can be certain that SEV is available and active on the remote host.

2.1.2. Iteration Two: SEV-ES

Although encrypting data in-use is a huge increase in security, some exploits could still be found. One of those is regarding the register states of a VM after it is halted. SEV is preventing the hypervisor from accessing a VM's private data, however it is still the hypervisor and host OS that have the full control over physical resource management. Consequently, a mechanism called Automatic VM Exit (AE) exists. Such an event is triggered from outside the VM. This essentially pauses it completely and transfers the control to the hypervisor. This pausing implies that the guest needs to be able to save its own internal state (i.e. register contents), otherwise it could not be resumed later on. The hypervisor could then read this written-out state of the VMs it was running, compromising security once again. For that reason SEV-ES [7] introduced a hardware mechanism to encrypt the state of a VM before it is being saved upon an AE.

On the other hand, sometimes the guest might want to exit on its own in order to communicate with the hypervisor (e.g. an emulation task). Events like that are called Non-Automatic Exits (NAE). In that case the VM might not want to have its entire state encrypted, since some registers may contain data that is intended for the hypervisor. Whenever the guest triggers an NAE, SEV-ES will not directly encrypt the VM state and hand off control to the hypervisor immediately. Instead, it sends an interrupt to the VM, that it can handle as it pleases. Typically, this involves publishing certain data to the hypervisor and then triggering the actual control transfer.

This requires a new way of exchanging state information with the hypervisor called Guest Hypervisor Communication Block (GHCB) [15]. The GHCB is a standardized data structure that has fields for certain registers (RAX, RBX, ...), information about the exit (e.g. certain parameters for the hypervisor), a bitmap that indicates which fields are present (e.g. is RAX set or not?) and some meta information amongst a few other fields. The guest can communicate data about the GHCB to and request actions from the hypervisor via a control register (MSR).

An example for issuing a NAE would be requesting to write to the debug control register. The guest should not try to access the register directly but rather ask the hypervisor to do that. The interrupt handler in the guest OS will set the necessary values in the GHCB. An exit code, in this case $0x37$, states the NAE type. The exit information will be given, stating which GPR should be used and possibly stores other information about the specific hypervisor action. Then, RAX will be set to the value to write. After everything has been written, the guest calls VMGEXIT, which will trigger the typical SEV-ES mechanism to encrypt the current VM state. The unencrypted GHCB however, will be accessed by the hypervisor. After the hypervisor completes the requested action, it can write data to the same GHCB. Thus, the GHCB allows for the hypervisor to send data as a response to the VM's request. The example above would not need any information from the hypervisor, thus no specific value will be written back to the GHCB. The hypervisor can return control to the VM, making SEV-ES decrypt and load the previous state and the guest will continue with the instruction after the

exit call in the interrupt handler. There it will copy data from the GHCB, i.e. the HV response, as necessary and continue its regular execution.

In summary, SEV-ES hardens the security model even more by encrypting everything the hypervisor could read in plaintext and making hypervisor communication much more explicit. Of course, the new features are also added to the attestation report, so the guest can verify whether SEV-ES is enabled. The guest OS needs additional modifications, however. It needs to implement the whole routine for handling the new kind of interrupt, including how to use the GHCB. The GHCB use cases are further extended for additional communication introduced by the next iteration.

2.1.3. Iteration Three: SEV-SNP

Lastly, a lot of important work has been done implementing SEV-SNP [16]. It is the key for enabling SVSM (section 2.2). Some key concepts SEV-SNP adds to the world of confidential computing on AMD hardware are as follows:

- Harden the VMs even more against malicious hypervisors by protecting integrity
- Introducing privilege levels within a VM
- Establishing a secure channel between guest and firmware
- Securely handle interrupts
- Fix known issues like certain side channels

The first three items in the list are most important for this work. Integrity protection is introduced by two new concepts: the Reverse Map Table (RMP) and Page Validation.

Page Tables, as will be later shown in section 2.3, map a virtual address to a (emulated) physical one. They are part of the operating system's memory abstraction towards user space programs. When memory at a virtual address is accessed, a translation unit will calculate the physical address and instead read or write memory from/to that translated address. As was shown before, SEV uses a special bit in the physical address in a page table to determine if the memory at the requested position is marked private. If so, it will automatically encrypt all writes and decrypt all reads. It also disallows direct access to that location from within the VM.

The hypervisor however, does not get restricted by the hardware from accessing the same physical address in memory. After all, it does have the full control over resource management. Confidentiality of the VM is protected; even the HV could only read the encrypted gibberish produced by SEV. Yet, the HV could write to that physical page in memory, either corrupting data or replaying data to the guest. Memory accesses from outside the guest obviously don't use the guest's page table and SEV does not store which physical page currently belongs to which VM. Thus, SEV cannot prevent any uncontrolled access to the encrypted page.

That is exactly the issue the RMP tries to fix. It maps the ownership of a physical address and enforces access control according to this mapping. The RMP will contain either an indicator of HV ownership or the corresponding guest physical address of the VM that owns it (it could also be owned by the CPU, but that is not of concern here). If the HV (write) accesses a page, it will only be allowed to do so by SEV-SNP if the indicator maps to it accordingly. In the case that a VM tries to access a page, the mapped guest physical address(gPA) from the RMP will be compared to the initially used gPA for address translation. If it doesn't match, the access will be denied. Not only does this map a physical page to a certain VM, but it also accurately maps it to a single page within the guest, as shown in Figure 2.1. This way another kind of attack is prevented, that would lead to data corruption due to multiple gPA mappings to a single system physical address (sPA).

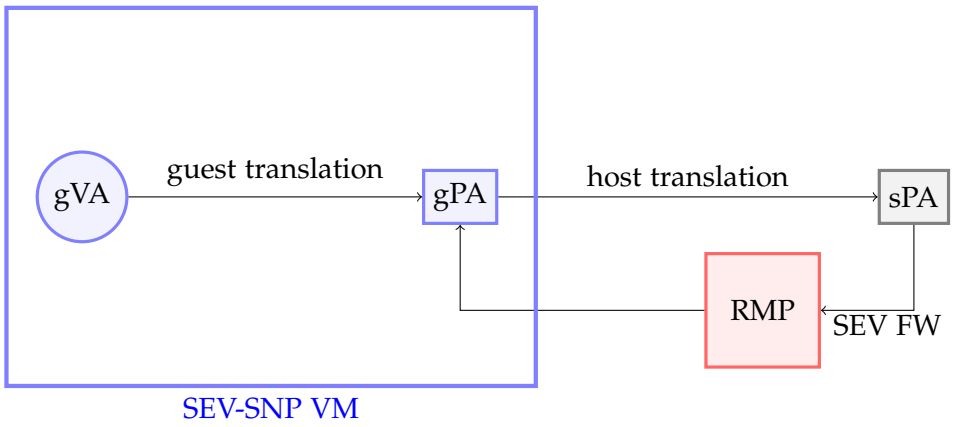

Figure 2.1.: Address translation from a guest virtual address (gVA) to a system physical address (sPA). The guest physical address (gPA) will be translated to a sPA outside the trusted VM; the result will be checked by SEV Firmware (FW) to contain the correct gPA.

The HV can manipulate the RMP only via a specified interface. This way it can fulfill its benign tasks like preparing physical pages for the use of a selected VM. However, it can never directly access the data structure, preventing any software bypasses the HV would attempt.

SEV is vulnerable to another mapping attack, however, that the simple RMP does not counter. That is the mapping of a single guest page to multiple physical addresses on the system. Although the mapping of a virtual address within the guest to a gPA is controlled by the guest itself (and can trigger SEV to encrypt data), the mapping of the gPA to the sPA is controlled by the HV. Thus, the HV could easily remap one gPA to a different sPA without alerting the guest. This would lead to inconsistencies within said guest VM.

To prevent this, the RMP uses an additional flag: a validated bit. A guest will request validation from SEV-SNP before using a page in memory. This will lead to the validated bit being set in its RMP entry. As explained before, the HV cannot access the RMP directly and has no way of setting the validated bit on its own. If it tried to change a mapping, the new RMP entry would have its validation bit set to 0, no matter what the previous mapping entry contained. Were the guest to access a page that has not been validated, SEV-SNP will throw an interruption to the guest, which would then know that the mapping has been tempered with. This kind of validation needs the guest to be aware of the mechanism. It can use the GHCB to request validation (amongst other things). Additionally, it is responsible for keeping validations consistent. It would be allowed for the guest to respond to an interrupt due to missing validation to just issue a validation request. It needs to know on its own whether this is safe.

So far, all measures taken to create a secure computing context for remote VMs have focused on isolating the guest from the untrusted outside world, living on the same hardware. SEV-SNP's Virtual Machine Privilege Levels (VMPL) take a completely different consideration into account. They are hierarchical levels of hardware isolation *within* a VM. A prime use case for that will be described in section 2.2. Implications for our security model are as follows. A kernel running at a certain privilege level can adjust access rights of a mapped page only for lower privilege levels than itself. These access permissions as well as the authentication are integrated into the RMP mechanism. A guest can emit an exit event (using the GHCB [15]), asking the hypervisor to run a different privilege level in the current VM. The guest is not forced to use this mechanism but especially under the aspect of cloud computing, this can add another security layer as described later.

SEV-SNP also offers a new request API (SNP_GUEST_REQUEST API [14]). It can be invoked by a special NAE, setting two valid gPAs in the GHCB. Both gPAs must point to shared pages (i.e. nothing encrypted by SEV), since those pages will be used for communication with the SEV-SNP firmware. However, the firmware equips the guest with keys upon guest launch to create a secure channel for the request mechanism. One page will hold a data structure containing a request, the other will be used by the firmware to write a response. Such requests could be asking to prepare for migration of the guest or, as will be useful for this thesis, getting an attestation report. Now, even communication with the firmware can be secured from the HV. The keys are stored in something called a "secrets page" that is stored in encrypted VM memory.

All of this shows that the guest OS needs even more adjustments in order to work together with SEV-SNP. At the very least it needs to validate its pages and implement a policy for what to do when an invalidated page is accessed (i.e. when to (re)validate a page). It can also implement anything to make use of the new VMPL mechanisms as well as the new SNP_GUEST_REQUEST interface. At last, it should be mentioned that some SEV-SNP features won't be explained, since they are not necessary for understanding this work. Different use cases are showcased in Figure 2.2.

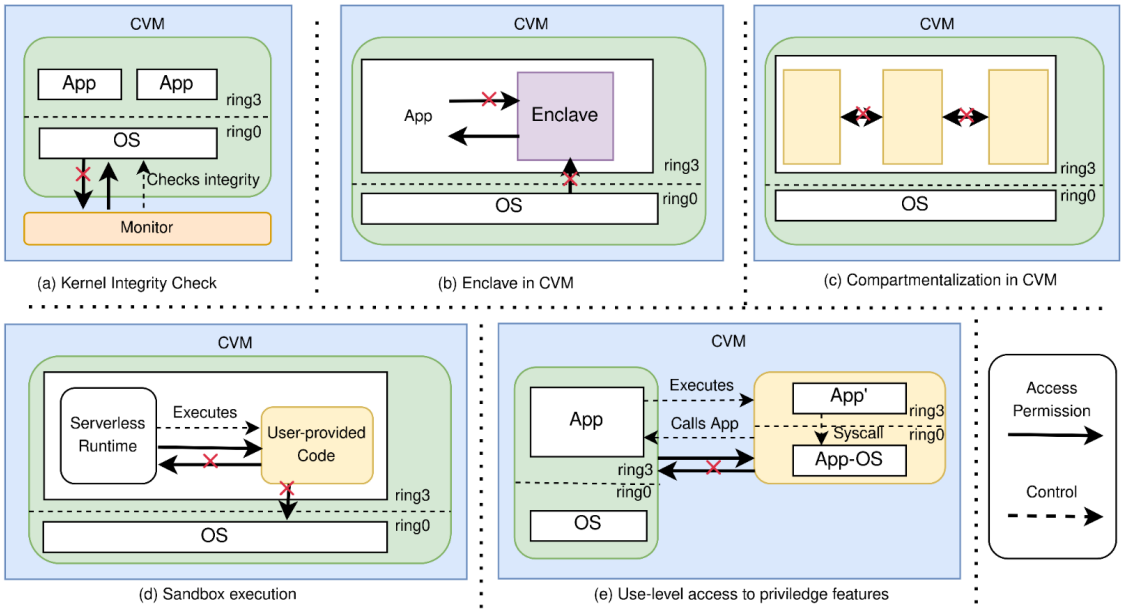

Figure 2.2.: Overview of different use cases for SEV-SNP and the newly introduced VMPL mechanism (courtesy of TUM DSE)

2.1.4. Notes on using SEV-SNP in this context

SEV-SNP forms the foundation for securing the VM and verifying the guest OS kernel. It will be assumed that the security mechanisms stated above are working as intended on the side of the CPU and cannot be exploited. This work only focuses on using stated mechanisms to further establish trust towards the remotely run machine. More on that topic can be found in chapter 3.

In this context the ability for separating the guest into different VMPLs is heavily utilized. Ultimately the verification uses the attestation command of SNP requests. Custom data can be sent to the firmware using the secure channel and the firmware will sign it (among other information stated earlier). This will be explained in more detail in chapter 5.

2.2. Secure Virtual Machine Service Module

The introduction of VMPL enables the development of entirely new use cases for SEV within remotely run VMs. An OS kernel can create lower privilege enclaves and control their use of SEV-SNP capabilities. This new enclave could be as isolated from the outside world (hypervisor and other VMs) as the higher-level OS wants it to be (at least in terms of exposing certain secrets). If the software running within this lower-level security would start acting maliciously or has some exploitable vulnerabilities (known or unknown to the guest owner), it might leak some information in shared pages, however.

Table 2.1.: Summary of features of the different iterations of SEV. Some key features are left out for SEV-SNP because they are not relevant in this context

Iteration	Features
SEV	• Encrypt physical pages in memory • Offer attestation
SEV-ES	• Encrypt register states • Controlled communication with hypervisor
SEV-SNP	• Integrity protection of memory pages • Privilege levels inside VM + inter-level communication mechanism • Flexible attestation • Secure interrupt handling • ... and more

By using different VMPLs it could not leak any private data from higher privilege levels. Even in case of privilege escalation within the enclave, SEV-SNP ensures through hardware that the execution contexts of the different levels stay separated. The only thing a lower-level enclave can do to influence higher levels is to write into shared pages or generally pages it has access to and request the hypervisor to run a different VMPL. This can be used to implement an interface between different VMPLs, but the higher level VMPL can decide how to react to the initiative by lower-level enclaves. It stays in control of its private data at all times. Only a higher level VMPL could control the lower-level page access rights. The higher level could then also verify certain security features of the lower level and can be used as a single access point to the firmware API, as will be explained in the following.

AMD envisioned one scenario of using VMPLs to increase resilience of confidential VMs. The idea is to load a security monitor, called SVSM, at the highest privilege level (numerically it is the lowest, VMPL0). This monitor would ensure the whole SEV-SNP context: it would get attestation and confirm SEV-SNP is active, it would validate pages and so on. After everything is set up, it would load and boot an actual general-purpose OS at a lower privilege level. Hence, a relatively small and manageable program would have the sovereignty over the context within the guest VM. The potentially vulnerable OS that is used to execute the designated tasks of the guest on the other hand, is limited in the ways it can exploit SEV features. SVSM also implements an interface that the guest OS can use to communicate with the security monitor and/or the firmware. The following section will lay out some details.

2.2.1. AMD's specification

AMD specifies crucial behavior and an interface of SVSM [13] to be adapted as a standard for the development of confidential VM images. As already described, the SVSM is

responsible for establishing the secure context and will load the guest OS at a lower privilege level than itself. It will then prevent the guest OS from directly communicating with SEV-SNP firmware. Instead, all tasks involving action from the firmware need to be delegated to SVSM.

The interface that allows that needs to be secured from the HV in a couple of ways. In order to do that, data that traverses VMPL boundaries need to do so in encrypted pages, that both levels can access. In order for the guest OS to discover if SVSM is present, and if so, where to put data for communication, the secrets page is utilized. Remember, this is where the firmware publishes keys to the guest for their secure channel. SVSM will grant the guest access to this secrets page, so it will need to make sure these keys get removed from there before the guest is booted. It can securely store them in any other page it has an exclusive access right to (i.e. encrypted pages for VMPL0). The guest OS can only read zeroed entries instead. However, SVSM uses its own fields in the secrets page. It stores some meta information that is relevant for the guest, like the memory range that is reserved for VMPL0.

It will also hold a gPA to a page that contains a new data structure: the SVSM Calling Area (SVSM_CAA). The most important purpose of the CAA is to detect whether a call was legitimately issued. Since the SVSM will always check for a call once the HV starts executing VMPL0, the HV could just try to make the SVSM execute something at random. When the guest actually calls for the SVSM to execute a command, it will write SVSM_CALL_PENDING to the CAA (and the HV does not have access to that). In the case that this field is not set, SVSM won't do anything except return to the guest OS. If SVSM deems the call to be legit, it will clear that field after handling the request and before returning to the guest. This enables the guest to do similar checks for wrongful HV intervention: if the field is not cleared when control is returned to the guest, the HV might not have executed the requested VMPL0.

It is worth noting, that this still means the HV can execute certain denial of service (DoS) attacks; but that is a general problem of remote VMs, with or without SEV features enabled. The important part is, that the HV can't inject malicious calls or responses.

The call itself utilizes registers and the SEV-ES mechanism that encrypts them. When the guest triggers a NAE, requesting the HV to run a different VMPL, its registers will be securely saved. Then the HV takes over and executes the VM at the desired VMPL, which will then reload the registers. Thus, the current VMPL can see the register state the previous VMPL has set upon exit, without the HV being able to read or modify the contents. It follows that the registers can be used for parameters and return values between the guest OS and SVSM. In theory, additional data can be transferred between the levels by using the CAA: it has a lot of unused space (of the 4 KB page that is exclusively dedicated to the CAA, only 8 bytes are specified).

The most important register is RAX. It contains two numbers upon a valid SVSM call: a 32bit number that denotes the protocol and another 32bit number that specifies a

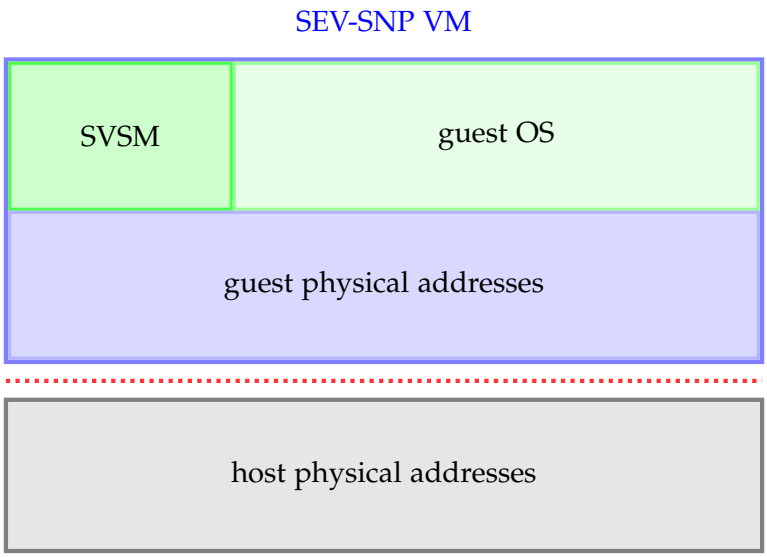

Figure 2.3.: The general layout of address space within the VM. The dotted lign denotes the uncertainty of the host mapping - it could be non-contiguous.

command within said protocol. Which other parameters are necessary (or if the CAA is used for additional parameters) depends on that specific command. RAX will also be used by SVSM to store a return code, but it could of course also use other registers to return certain values to the guest OS. The protocols separate different use cases and capabilities. The current specification (July 2023) contains three of them:

- Core: Mandatory, versioned protocol that defines some basic interaction with the firmware (like validation or creating new vCPUs) as well as some communication with SVSM
- Attestation: Optional, versioned protocol allowing the guest OS to request attestation of SEV-SNP services
- vTPM: Optional, versioned protocol requesting the SVSM to execute commands on a virtual Trusted Platform Module (TPM) [18]

For the work of this thesis, a new protocol will be added, as it defines a new and distinct use case of SVSM communication.

2.2.2. Coconut SVSM

CoconutSVSM [3] is an implementation of AMD's SVSM specification. It is a successor to Linux SVSM [1] which was developed directly by AMD (but is now deprecated). More features are still being developed, like different execution modes. The base of the SVSM spec, including the secure boot of SVSM, preparing and launching the guest

OS as well as the mechanism underlying communication via the CAA is present and functional. For the most part, the implementation uses Rust, trying to use its security and robustness features wherever possible. There is an active community of contributors that develop the core project. It has about 30k lines of code [1] which is about 0.1% the size of the Linux kernel, making a strong case for using CoconutSVSM in VMPL0 instead of running Linux "natively" in the VM.

The guest OS does not need to know which SVSM implementation is being used; CoconutSVSM adheres to the calling convention specified by AMD and implements the only required protocol, which is *core*. The OS can query the availability of protocols via a *core* protocol command or otherwise receive a `SVSM_ERR_UNSUPPORTED_PROTOCOL`. CoconutSVSM will be used as the base for this thesis' implementation.

2.3. The Linux Kernel

The guest OS needs to be modified in order to work with CoconutSVSM and the popular, open-source Linux kernel is a great choice for that. The kernel itself is huge and complex and far outside the scope of this thesis. This section will only briefly talk about some features that are being used for implementing the verification mechanism, especially regarding their implication for SVSM. These features are:

- Use of protection rings and access control
- Memory management in page tables and virtual memory
- Compilation of kernel modules

2.3.1. Security Mechanisms

Before VMPLs another hierarchical protection mechanism existed that separated more privileged programs from lesser privileged ones. These are called protection rings and are quite and integral part on Linux that realize a separation into user space (on ring 3) and kernel space (on ring 0). They can be used completely orthogonal to VMPLs [16], so the guest OS does not need to adjust on that account. Furthermore, only ring 0 is granted privilege to use the SVSM API. User space can only use interfaces offered by the kernel in order to get access to SVSM functionality, hence strengthening security within the guest. Also, other mechanisms that establishes security within Linux, like access control for file contents remain intact when used in a lower VMPL. Thus, the typical security model for Linux can still be applied.

[1] Measured via `https://codetabs.com/count-loc/count-loc-online.html`

2.3.2. Memory

As the goal for this work is to protect kernel memory, it is worth looking at memory management on Linux. Once the kernel is fully booted, it will have set up its page tables [17]. From then onward, addresses will mostly be given as virtual addresses. For an actual memory access, such a virtual address will be translated in several stages using the kernel's page tables.

One implication of that is affecting the user space; each user space process will have its full range of virtual memory and only the kernel decides, where the physical pages are located. The actual translation process is supported by hardware (this is how SEV knows about the C-bit for instance). The guest kernel will treat the emulated memory from the VM as if it were actual hardware addresses. However, the physical address from the viewpoint of the Linux kernel is exactly what was described as a gPA earlier in the context of virtualization.

Linux and SVSM share the same guest physical address space(see Figure 2.3); for Linux it looks like some other system occupies the same hardware (memory) and that system's memory region is reserved [13]. It's quite convenient since a physical address within the Linux kernel running e.g. in VMPL2 and a physical address within SVSM at VMPL0 will be identity mapped. That's how Linux can request validation for a certain page by stating a physical address (a gPA). SVSM will read the address exactly as intended and can validate the page the guest OS wants to have validated on its behalf.

2.3.3. Kernel Modules

A very useful mechanism of extending the kernel for custom tasks is by using kernel modules. Code within a kernel module can link against kernel libraries (but therefore not against user space libraries like glibc). Such a module can be loaded dynamically or compiled into the kernel. When loaded, it will also share its virtual address space with the rest of the kernel.

A typical use case for kernel modules are device drivers, but they are not restricted to it. On the other hand, *device files* can be used to implement user space interfaces. Despite their name, they don't need to correspond to a physical device (but they can, of course). The kernel module can define a new type of (virtual) device (using a currently unused identifier) and a corresponding device file can be registered in the file system. It's the module's responsibility to handle the file system actions that are called on this device file. This could be simple actions like a read or write, as are typically known for regular files.

The file system offers another action that allows for more complex use cases called `ioctl`. Within an ioctl action, custom commands can be defined that allow for designing more flexible interfaces between a kernel module and user space programs. This work implements the verification mechanism using such a module and offer functionality to the user space for obtaining the present measurements and dynamically requesting more measurements.

3. Threat Model and Security Considerations

No system can ever guarantee 100% security, but it can be designed to be secure against a certain set of attack vectors. This chapter will contain a brief discussion on which attacks are considered for this work and which aren't. It will also shine light on the bigger picture, i.e. how the new memory verification mechanism can contribute to the security of and trust in a remotely executed confidential VM. Later, chapter 5 will refer to some considerations when explaining details of the code.

3.1. Platform Security

The main security components are the platforms being used, namely SEV-SNP and CoconutSVSM. This work's security can only be regarded under the terms of the security of those platforms. If e.g. SEV's encryption mechanism were to fail, no security could be established. The SEV white papers specify their security models, arguing what attack vectors they are secure against, and state how they strengthened security of previous iterations [8], [7], [16]. They rely on typical security assumptions, like the security of well known cryptography primitives and on assumptions about an attackers capabilities. They grant an attacker quite extensive control over the system; only the AMD hardware + firmware and the current SEV-SNP enabled VM are considered as trusted components.

Everything else could act maliciously or be attacked without compromising the computation context of the guest. This includes software based attacks from or against the hypervisor and even some hardware attacks e.g. cold boot attacks against physical memory modules. However, even in the latest SEV-SNP version, some attacks are not in the scope of security. This includes DoS attacks against guest VMs and quite a few, rather complex side channel attacks (The SEV-SNP white paper provides a handy overview table for security against certain threats).

On a second note, there is still some remaining doubt about how well SEV-SNP does actually fulfill its security goals. In the past some very elaborate and creative side channel attacks have emerged once in a while for even established systems. Some examples would be Pegasus [2], which was a zero-click exploit on Apple devices, allowing an attacker to basically get full access to the system, or Spectre [9] that used exploits in hardware branch-prediction to read confidential data in CPU caches. More recently, an attack was executed on XZ utils [5], opening a backdoor via the very popular Unix utility present in most Linux servers and desktops. These attacks took the cybersecurity

world by surprise and show very impressively, how quickly certain assumptions about security can be disproven. SEV is a relatively new set of features and a lot of software utilizing its mechanisms is not fully developed yet. For the near future, users need to be on the lookout for newly discovered side channels.

In fact, researchers are already on the task of testing SEV-SNP's security concept to the bone. One feature that wasn't discussed in this thesis is the handling of interrupt injection in SEV-SNP. Its aim is to prevent a malicious hypervisor of generating interrupts for the guest OS at times when it wouldn't expect them. Software utilizing SEV-SNP seems to not have full support for these features yet. Systems are vulnerable to the WeSee attack [11] if such protection fo interrupts is not active. It constructs certain interrupts that the guest's SEV-ES interrupt handler, that lead the VM to leak memory.

Another attack creating ciphertext side channel [10] in order to compromise all confidential data within the guest appears to be working even when SEV-SNP restricts the HVs ability to inject interrupts. This timing based attack pauses the VM through page faults that will trigger the guest to call the HV to handle it. The HV will simply not handle the error and return to the guest anyway. The guest can't resume its usual execution and will eventually perform a context switch. The researchers have devised a method to detect this context switch (even with an intermediary like SVSM), locate certain types of data in memory and use it to break cryptography in the VM.

Even though these vulnerabilities are known, for the context of this thesis SEV-SNP as well as the base implementation of CoconutSVSM are assumed to be secure. No attempts will be made to fix these issues or to protect the mechanism against these attacks. Once SEV-SNP is secured against all known vulnerabilities, this mechanism may be secure under the given assumptions.

3.2. Securing the Linux Kernel

The security goals of this thesis are as follows:

1. Securely measure the content of any page
2. Securely measure the contents of all kernel pages shortly after booting
3. Prevent the kernel from being tempered with
4. Verify the measurements

The first goal is realized using SEV-SNP's attestation mechanism and SVSM as a secure interface to that. A digest of the page contents is generated from within SVSM and inserted into the attestation request as user data. SEV-SNP will sign the user data without changing it. The guest OS can read the report and find the signed digest.

The second goal is and extension of the first. Instead of an arbitrary page, the kernel pages need to be enumerated and a digest needs to be generated within SVSM again. It needs to be ensured this happens shortly after booting.

The third goal goes hand in hand with the second. If the kernel ensured to be read only, then measured, it is guaranteed that the kernel matching the generated digest is run at all times (until shutdown).

Of course, the signature and digests also need to be verified. This can happen remotely and in user space and is not directly a feature for the confidential VM.

The VM needs to start in an unencrypted state, thus the HV could try to load a maliciously modified Linux kernel into the guest image. Using this model, such an attack would be detected. The digest would not match or would not be sent at all. In order to avoid replays, the SVSM also needs to generate a nonce to append to the digest and publish it to the guest OS in plaintext. A malicious guest OS could drop the nonce, so that it could not be verified in the signed digest; the VM owner would then assume no trust can be established and would not upload any data. Since the kernel would be in read-only mode, the measurement would be repeatable during runtime.

4. Design and Architecture

With the goals for the adjustments being specified, there are two components that need to be adjusted in order to achieve them.

One is the security monitor that has been introduced in section 2.2, the Coconut SVSM and the SVSM specification in general. Although it might be possible to exploit already defined protocols, such as the attestation protocol [13], that would be rather inelegant and generally against all engineering principles[1]. This thesis instead implements a separate protocol as a way to expose a dedicated API to the guest OS. Said protocol will be defined in subsection 4.1.1 and then adapted by a modified version of Coconut SVSM.

The other component is the guest OS itself; it will need to consume the newly created API in order to fulfill the tasks. The Linux kernel should offer a way to interact with Coconut SVSM for the sake of verifying memory content. Utilizing that API comes with great flexibility. For instance, kernel modules that are loaded after the initial boot or even some user space programs could thus be verified.

4.1. Extending SVSM

4.1.1. The Check Protocol

Similarly to the original SVSM specification published by AMD [13], the new protocol is defined and described regarding the new capabilities added. The rest of that specification will still hold true, e.g. an SVSM module is still designed to run at VMPL0. When it comes to the definition of protocols, the specification is easy to extend. For reference, Figure 4.1 visualizes the details of the process introduced in section 2.2. The important part here is the content of RAX. It is separated into two parts: the protocol ID and the ID of the function within that protocol.

Hence, the steps in defining the new protocol are:

1. Reserve an unused number as the ID for the new protocol
2. Define functionality of new protocol

[1] It would resemble a "quick-and-dirty" approach where changing the functionality of one component (e.g. the attestation protocol) would impact or even break another, semantically distinct component. That affects readability, maintainability, testability, separation of concerns, and makes for a lower quality architecture.

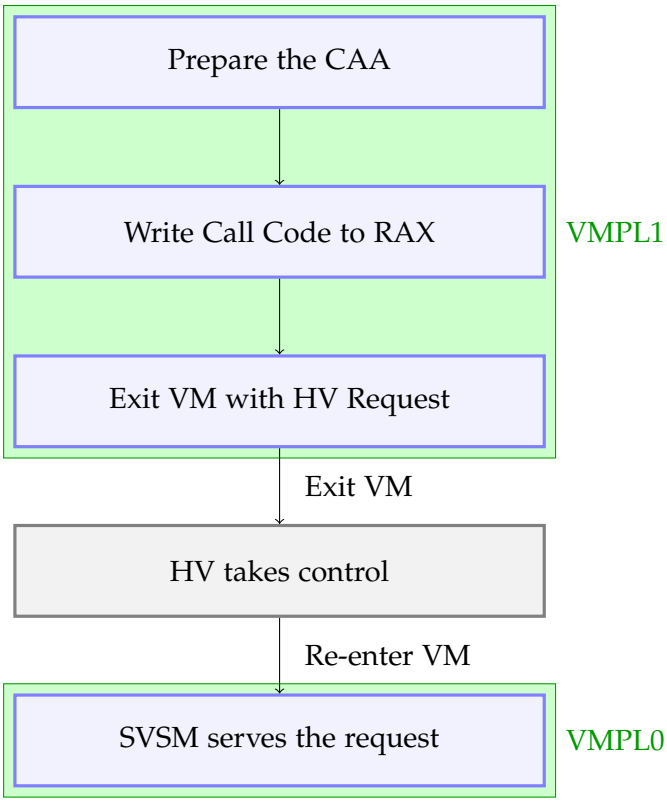

Figure 4.1.: Control flow of a call to SVSM. The guest OS first prepares everything, then exits to the HV, which in turn is supposed to return control to the VM. The actual call is denoted by the value in RAX. The guest OS naturally runs at a VMPL>0 (usually 1). The Hypervisor needs to return control to the VM at VMPL0, which will run the SVSM module. The CAA can be used to verify the Hypervisors actions: SVSM can check whether it is indeed being called on behalf of the guest OS; the guest OS can check whether SVSM has run the request (potentially unsuccessfully), i.e. the HV actually started the SVSM.

Table 4.1.: Overview of services of the Check Protocol

Call ID	Name	Function
0	SVSM_CHECK_SINGLE	Check a single page in memory regarding certain criteria
1	SVSM_HASH_SINGLE	Create a digest of a single page in memory and return it
2	SVSM_CHECK_RANGE	Check a range of pages memory regarding certain criteria
3	SVSM_HASH_RANGE	Create a digest of a range of pages in memory and return it
4	SVSM_CHECK_MULTI	Check a list of pages in memory regarding certain criteria
5	SVSM_HASH_MULTI	Create a digest of a list of pages in memory and return it

3. Assign ID to each function of the new protocol

In terms of the architectural extension of Coconut SVSM that means, a new module needs to be created that contains all the desired functionality. Its only public interface is a function that accepts a numerical value. This dispatcher function should then execute one of the module's specific functions, according to the numerical value (i.e. the function id, meaning the second half of RAX's value). The global dispatcher in SVSM, which executes a protocol module's public function depending on the protocol id (the first half of RAX's value), only needs to be extended so that it can call the new module's dispatcher function when the protocol ID matches.

The proposed protocol was given the name *Check Protocol* as it checks memory regions on behalf of the user. The ID is arbitrarily chosen to be 3, the next unused value by the original SVSM specification. Multiple services are exposed to address different needs and increase flexibility. The following sections give detailed explanations of these services; Table 4.1 shows an overview.

SVSM_HASH_SINGLE

Generates a SHA512 hash of a single page in memory. The address is to be given as a guest physical address. Call Parameters are as follows:

rcx Page address to generate the hash from.

rdx Page size information.

r8 Guest physical address to write the report to.

r9 Size allocated for the report.

rax Protocol ID and call ID (1).

SVSM_HASH_RANGE

Generates an iterative hash over a specified contiguous range of pages. It receives an address as the first page to process and an amount of pages to iterate over. Parameters are as follows:

rcx Start address to generate the hash from.

rdx Page size information.

r8 Guest physical address to write the report to.

r9 Size allocated for the report.

r10 Amount of pages to hash.

rax Protocol ID and call ID (3).

SVSM_HASH_MULTI

Generates an iterative hash over a list of specified pages. It does so by similar mechanism of the core protocol's *SVSM_CORE_DEPOSIT_MEM* call. A guest physical address is given that contains a list data structure. This list enumerates all pages that are to be included in the hash. Parameters are as follows:

rcx Page address containing the list of pages.

rdx : Page size information.

r8 : Guest physical address to write the report to.

r9 : Size allocated for the report.

rax : Protocol ID and call ID (5).

The list data structure is constructed equivalently to Table 7 of the SVSM definition **??**.

4.2. Extending the Guest OS

The guest OS is extended by means of a kernel module since the SVSM interface is based on kernel space functions. It's job is to take some user input, then set the register values in the CAA and issue a proper VM exit, signaling the Hypervisor to execute the VM at VMPL 0.

The user interface is defined via a device file. This allows the file system to be leveraged. The simple API just uses read and write operations on said device file to parse inputs and return outputs. A more flexible API could use `ioctl` instead. Then, a fully fledged user space program could use the `ioctl` to provide a more usable interface from user space. However, the guest OS side is only supposed to be a small example that can be improved upon

The user space write to the device file simply contains whitespace separated values signifying the call parameters.

5. Implementation

So far, everything has been held at a fairly conceptual level. In order to form a full understanding, a look at the code base is necessary; this chapter is guiding through a few important parts of it. The following sections will provide some insight into the internals of the SVSM modifications, the kernel side and the interplay between guest OS and the security monitor.

Between the SVSM module and the Linux kernel there is not only a logical and spacial separation, but also a separation by language. CoconutSVSM is written mostly in Rust, while Linux is written in C. This also imposes C as a language of choice for writing kernel modules, as they can be seen as direct kernel extensions. An interesting restriction applies to both codebases, however, which is the absence of standard library support. One reason for this is, that standard library calls often abstract away system calls, which are pure user-space interfaces to the Linux kernel (which is not available in SVSM and a kernel module already is in kernel space). Another is, that certain things like memory management work differently in kernel space, and user space implementations won't do. As a consequence, pre-defined functionality is limited. For the Linux side, special headers from the linux kernel (some of which specifically intended for use by kernel modules) can be included. The SVSM needs to implement everything on its own; the CoconutSVSM project already does a great job at defining a lot of useful interfaces, and the TUM-DSE extensions reimplement e.g. `malloc` functions.

Firstly, the SVSM side of things will be presented with a focus on this work's contribution. Then the kernel module will be explained, as well as how to execute it.

5.1. SVSM Modifications

The modifications on the SVSM side mainly reside in two parts. One is the module for the new *check* protocol. The other is the creation of bindings for the cryptographic library.

The latter uses HACL[1] implementations under the hood. Since the protocol needs to hash pages iteratively, the stream API of its SHA512 implementation is used.

Rust offers a powerful foreign function interface mechanism that allows for the call of C functions within Rust (and the other way around). First, some wrappers for functions and helper types are implemented in C which are then compiled into a static library. The

[1] Documentation: `https://hacl-star.github.io/`

Rust compiler is instructed to link against this newly created library. Then a wrapper module in Rust defines bindings to this library; moreover it offers a safe interface.

When interfacing to C code, using raw pointers and generally doing things that Rust cannot guarantee to comply with its paradigms, unsafe Rust needs to be used. In order to offer a safe public interface, each interfaced C function is wrapped. This is done by implementing functions for a Rust struct.

The hashing stream API generally has three steps: an initialization step, to allocate and initialize everything that is needed, an update step, that computes a hash given the current state and a new value, and a finish state, which retrieves the final hash value. The update step can be called iteratively as long as the state is initialized and not yet destroyed.

For Rust it is quite natural to define such functions directly for a given struct. Hence, `HashState` is a struct that contains the current hash state. The init step generates a new instance of it. Various `hash_update` functions for receiving different types of values directly operate on a given `HashState`. The finish step frees all previously used resources, preventing memory leaks. Generally, function inputs are sanitized and error handling is added, such as to create actually safe Rust functions. The Hash digest itself is defined to be represented in memory as a C struct would, enabling passing references to instances in memory between C and Rust (and vice versa).

Figure 5.1 shows an example of how the safe wrapper has been constructed.

```
pub fn hash_update_slice(&self, sl: &[u8]) -> Result<(), SvsmError> {
    let chunk = sl.as_ptr();
    let raw_state = self as *const HashState;

    let err_code = unsafe { hash_update(raw_state, chunk, sl.len() as u32) };
    result_from_code(err_code)?;
    Ok(())
}
```

Figure 5.1.: The stream API update function for a slice of bytes. It prepares everything for calling the C code. This means it needs to retrieve the raw pointers to both the function argument (aka the value to be hashed) and the current HashState. This can be safely done as those variables are defined in safe Rust. Thus the values are guaranteed to be sanitized (i.e. initialized and all to specification). Thus the unsafe call to the C function is actually not unsafe. The `result_from_code` function is a helper function handling errors.

As for the protocol, the specified stream API for hashing the values can now be used like any other Rust module. The interface has already been shown in chapter 4. Figure 5.2 gives insight into the `hash_single` function. This is the easiest case and it can be seen how the parameters are being extracted, the hash API is used and the report

is being prepared. Thus it shows the core functionality of the protocol; the other calls merely extend this by iterating over multiple inputs.

5.2. The Kernel Module

For completeness sake a short look and explanation of the simple kernel module is given. The module has access to a struct called `svsm_call` that repressents the CAA. It also has access to an external function called `do_svsm_protocol` that initiates the VM exit. It registers file operations with a `file_operations` struct. User space data is retrieved and sent with special kernel functions. Figure 5.3 shows the relevant part of the module regarding the SVSM interface.

The use of the kernel module's interface (towards Linux userspace) is fairly simple. Once the module is loaded, there will be a device file at `/dev/svsm_client_dev`. The SVSM request is triggered once a value is written to this file. It expects one numeric value, the ID of which process the address range should be taken from, and then, separated by a whitespace, another numeric value that represents an address. This address is a virtual one; for testing purposes it can be obtained e.g. via `/proc/ID/maps` where ID is a process ID. The kernel module will transform this virtual address into a guest physical one by use of this ID. Thus, any memory page can be checked by SVSM with this kernel module.

The result can be immediately read from the same file (e.g. using `cat`). There is no waiting time, because the kernel module reads the digest from a previously specified memory address; the one SVSM has written to. The process of SVSM writing to that location is not visible by the guest kernel; The moment it resumes execution the result is available.

```
fn attest_hash_single(params: &mut RequestParams) -> Result<(),
    ↪ SvsmReqError> {
// Parameter checks and conversions
// ... left out for brevity. See caption for more information.

// Hashing

let mut digest = HashDigest::new();
let mut page_slice = unsafe { page_vaddr.to_slice::<u8>(page_size_bytes)
    ↪ };

let mut hasher = HashState::hash_init();
hasher.hash_update_slice(&mut page_slice)?;
hasher.hash_finish(&mut digest)?;

// Write back for guest OS

let mut exchange_buffer = [0u8; REPORT_RESPONSE_SIZE];

if digest.size > USER_DATA_SIZE {
    return Err(SvsmReqError::invalid_parameter());
}
for index in 0..digest.size {
    exchange_buffer[index] = digest.digest[index];
}

let res_paddr = res_addr.page_align();
let res_offset = res_addr.page_offset();
prepare_guest_report(&mut exchange_buffer, &mut res_paddr, &mut res_offset,
    ↪  page_size_bytes, valign);

Ok(())
```

Figure 5.2.: This function has three parts as the comments in the code suggest. The first part uses a bunch of helper functions to check input parameters for validity, and convert them to usable variables. They are forwarded to this function by the `RequestParams` struct. The hashing section demonstrates what has been described in this section earlier. Lastly, the result needs to be prepared. Therefore, the `prepare_guest_report` function of CoconutSVSM is used. This initiates the call to the firmware to generate the report with the provided user data (i.e. the hash). Then it writes said report to the given address. This is what the Linux guest kernel can later receive. After everything is wrapped up, SVSM automatically requests returning to executing the higher VMPL level.

```
/* make svsm call */
struct svsm_call check_call;
/* call id */
check_call.rax = SVSM_CALL_HASH_SINGLE;
/* physical address of page to check */
check_call.rcx = (u64) p_addr_phys;
/* page size used */
u64 page_size_indicator = 0;
if (PAGE_SIZE != PAGE_4K)
    page_size_indicator = 1;
check_call.rdx = page_size_indicator;
/* physical address of report buffer */
check_call.r8 = (u64) tmp_buf_phys;
/* size of report buffer */
check_call.r9 = (u64) BUF_LEN;
int ret = do_svsm_protocol(&check_call);
return (ssize_t) length;
}
```

Figure 5.3.: This excerpt shows the part that prepares the CAA, i.e. sets the call parameters, and then triggers the exit of the VM with the request to run at VMPL 0. In this case, the function call is hard coded. The parameters like `p_addr_phys` has previsously been parsed from the device file operations.

6. Evaluation and Measurements

In the following the performance of the virtual machine is measured and evaluated. First, a comparison between a VM that does not incorporate the security monitor is compared with one that does utilize SVSM. Before showing any results and interpreting them, the test environment and more concrete goals will be explained.

6.1. Test Environment

The tests are conducted on a machine that has an AMD EPYC 7713P CPU[1] (64 cores) and a reported 991 GB of total RAM available. This generation of EPYC processors natively offers the SEV-SNP features that we need for SVSM to work.

The virtual machines are instantiated through QEMU[2] and KVM; The QEMU version used supports the required SEV features. The virtualization uses QEMU's "EPYC-v4" CPU model and offers 8 GB of RAM as well as eight cores to each guest. Both, the machine that does directly boot into the Linux kernel and the ones that use SVSM, are configured to be as similar to each other as possible. That is to eliminate as many interfering factors to the measurements as possible, allowing for meaningful results. The VM not booting into SVSM will from hereon be referred to as *plain VM* and the ones that do as *confidential VM*; if need be it will be clarified if the confidential VM uses the modified version of SVSM. Both VMs run an Ubuntu 22.04 LTS cloud image[3] as the guest OS. The notable differences are as follows:

- The confidential VM uses the *sev-snp-guest* object that QEMU's interface offers to load the SVSM image; The plain VM naturally doesn't
- The confidential VM uses an SVSM image that uses *EDK 2* firmware[4] (an open source UEFI firmware project) to boot the guest kernel from within the SVSM kernel; the plain VM uses the *EDK 2* firmware directly by means of QEMU pflash devices
- The confidential VM needs a modified Linux kernel; as discussed in chapter 2, the guest kernel needs to be aware of SVSM being in use, hence this measure is unavoidable

[1] Specs: `https://www.techpowerup.com/cpu-specs/epyc-7713p.c2375`
[2] `https://www.qemu.org/`
[3] Images pulled from here: `https://cloud-images.ubuntu.com/jammy/`
[4] Base Repository: `https://github.com/tianocore/edk2`

The UnixBench benchmark tool[5] is being used. It is a general-purpose tool to measure POSIX overhead, file read/writes/copies, execl throughput, arithmetic performance and more. Summarizing the tests, it then generates scores and normalizes them against a certain baseline, to make results more easily comparable.

However, it is not necessarily optimized for the use in VMs and some tests might be too trivial to be of real world significance.

Other benchmark tools will be used to understand the performance regarding the employed virtualization mechanisms more directly. These are not full-system tests; i.e. they don't initialize large workloads and then measure the overall performance on them. Instead they are more fine-grained and only measure certain aspects. Further details will be explained in subsection 6.3.2. The base for these tests comes from TUM DSE's CVM_eval project[6].

6.2. Evaluation Goals

It is important to specify what is in the scope of this performance evaluation and what isn't.

Firstly, a valid concern might be that utilizing SEV, i.e. the encryption of every write to memory and decryption of every read, might itself impose a significant overhead to the running VM. However, a research team at SUSE has conducted extensive experiments in that regard [4]. They used a different OS distribution than is used in this work, and they also had older EPYC chips in use. Their modified kernel directly used the SEV interface, without a mediator in between, mostly making sure that all in-memory data is encrypted. They came to the conclusion, that the performance impact is marginal compared to a VM that does not use SEV features at all. Thus, it is assumed that the hardware accelerated encryption mechanism is fast enough as to not add a notable overhead and its performance impact is not further investigated as a singled out metric.

This work focuses on the performance impact of specifically running Coconut SVSM as a monitor in the VM. Not only will its addition delay the VM startup, since the Linux guest OS will be booted from within a fully initialized SVSM, but any communication between Linux and SVSM requires two context switches **to** the hypervisor and two switches **from** the hypervisor per message. The question to be answered is, how this affects the performance of certain types of workloads in the guest.

Both, the plain VM and SVSM VM system benchmarks are conducted for several scenarios, concerning the capacity of the host. More precisely, a test run is conducted in a case where only one VM is started on the host, then four in parallel and lastly eight in parallel. For each of the parallel runs, the same type of VM is used (plain vs. SVSM) and the same workload is started simultaneously. There is no synchronisation between

[5]Originally, the BYTE UNIX benchmark suite (from BYTE MAGAZINE) the updated version of this tool is now available from this repository: `https://github.com/kdlucas/byte-unixbench`

[6]The CVM_eval repository: `https://github.com/TUM-DSE/CVM_eval`

benchmarks, but since the test iterations are sufficiently high, each parallel VM will be running the same test (and thus the same kind of workload) as all the others for the majority of the execution time. The case of eight parallel VMs is in theory fully utilizing all the host's processing cores. Setting these different scenarios yields some more insight into individual behavior on multi-tenant architectures.

Apart from these full-system tests that are neither aware of the virtualization, nor of SVSM utilization, some tests are being conducted to measure more specific events. For instance, an interesting question is how much SVSM impacts boot-up time for VMs. This can be especially a concern for huge commercial cloud providers that need to run large amounts of VMs by various customers. A lot of VMs need to be run and started up in parallel, and slower boot-times may be a severe trade-off for the cloud provider. The resulting added security for the customers need to be weighed against the flexibility of the whole system. In order to measure such things, small adjustments need to be made on the side of the guests, and the host runs BPF programs. Both will be explained later.

6.3. Benchmark Execution

The next two sections go into detail of what kind of tests have been run, and their purpose concerning the previously stated goals are drafted. Moreover, the results are presented or referred to and interpreted. Firstly, the conducted UnixBench tests are shown, then the selected benchmarks from CVM_eval are looked at in more depth.

6.3.1. Unixbench

The following full-system benchmarks have been run in both VMs, the plain VM as well as the confidential VM (with the unmodified SVSM; the added protocol is not called during normal operation and thus cannot induce performance overhead in those scenarios):

- UnixBench system tests
- UnixBench misc tests (with the exception of two upstream broken ones that are included by default)

All UnixBench tests have been set to 100 iterations each via its `-i` option. Each of these test runs has been executed with one and then with eight concurrent executions (matching the eight cores of each VM). The test scenarios take multi-tenancy into account; the system tests have been executed with one, four and eight concurrently running VMs of the same type, all running the same benchmark. Although the specific test instances of each test run are not fully synchronized, since each test runs for a considerable amount of time, for the most part of each test all machines run similar workloads simultaneously. This showcases possible impact of multi-tenancy on the otherwise isolated VMs in

the case that several of these machines need extensive and similar access to the same hardware resources (and/or resources provided by the HV or orchestration software).

The following should give some insight into what benchmarks have been included. They are grouped logically, not necessarily in the same way they are grouped in the UnixBench application. For each group it will be stated what instance they belong to, what tests are included and what their purpose is.

Process related. Part of the UnixBench system tests. This includes tests for pipe throughput (one process writing to and reading from the same pipe), pipe-based context switching (two processes constantly communicating via two pipes), process spawning (repeatedly spawning and killing processes), execl throughput (repeated process image replacement with `exec()` function family calls) and system call overhead. They generally are supposed to measure performance of typical operations necessary for creating (new) processes, inter-process communication and communication with the system.

Program execution. Part of the UnixBench system tests. Includes benchmarks that simulate typical C code as well as shell scripts. It is questionable how well these benchmarks capture the "reals world" in their simulation; especially the C programs are synthetic and quite old, hence might not represent anything modern or relevant at all. Nonetheless, they comprise a more varied workload than a lot of the other UnixBench tests.

File system usage. Part of the UnixBench system tests. Consists of different benchmarks for reading, writing and copying files using various buffer and file sizes. Generally measures the performance of interactions with the file system and disk access overall.

Arithmetics. A bunch of arithmetic tests for integer and floating point operations in the UnixBench misc benchmark. Only conducted for the case of single VM execution. These are less interesting and only executed during the single VM runs.

Others. Two more tests from the UnixBench misc benchmark. One is a test running a recursive program (Tower of Hanoi) to test performance of such programs with fast-growing stacks. The other is a performance test of using the "grep" utility on large files, thus tests specifically the scenario of string-matching with interaction of the file system. Both tests are only executed during the single VM runs.

A full overview of the results can be found at Appendix A; The tables there show the absolute results for all combinations of scenarios: Two types of VMs (plain and confidential), two modes of intra-VM concurrency (one and eight concurrent test executions) and three modes of system concurrency (one, four and eight concurrent VMs). Each value is an average over a total 100 test iterations in the specified scenario. This is mostly to tackle general variance that can occur for very few iterations but also to

capture performance over a longer timeframe. The execution of all system tests took around 6+ hours per scenario.

It would be expected that performance would monotonously decrease the more VMs are run in parallel under the workload. Interestingly, this is not always the case in the measured results. For the single process case, single VM runs in both scenarios, plain and confidential, performance was actually significantly lower than in the four and eight parallel VM runs. It is difficult to pinpoint an exact reason for this outcome. It could be a fluke in the experiment, like accidental maintenance work conducted on the host at the same time that impaired VM performance. It could also be another issue with host scheduling or the hardware like clock adjustments due to heat issues (although this seems to be unlikely, since the other scenarios use much more of the CPU and hence should have it generate more heat). It is impossible to investigate this issue in retrospect with only the output recorded from UnixBench. As a consequence, from here on, only the scenarios running the tests in parallel within the VMs are considered. Luckily, these scenarios are the most relevant, since they fully leverage the allotted hardware. In real world scenarios, applications will run concurrently. This is partly due to performance reasons, but also because customers pay for the resources that they reserve; consequently, they want to utilize those resources as efficiently as possible to save costs.

Comparing arithmetic performance, the confidential VM performs insignificantly worse (i.e. the integer test is about 0.003% slower in the confidential VM). This could be attributed to random noise or very occasional context switches that are unrelated to the compute task. Among those tests that have only been run in the single VM scenarios, the benchmark grepping in a large file is the odd one out; its performance drops by nearly 30% on the confidential VM.

Looking at the remaining benchmarks, the expected behavior of decreased performance in relation to the amount of parallel VMs can be observed for most of them. For some cases these performance drops are quite significant. However, in the confidential VM case, again some performance increases have been detected going from the single VM to the four-in-parallel VM case. Regarding the file copy benchmarks, this phenomenon is significant (around 8% for the largest file copy benchmark). Perhaps some kind of overhead is introduced by SVSM regarding disk access that is mitigated by shared use of HV resources. Once again however, it is not possible to make a certain assessment of the issue.

Lastly, a small selection of tests is compared against each other. Namely, the largest file copy (using a 4kB buffer), the process creation and the two shell script benchmarks will be investigated further. The first two are chosen because they represent heavy-duty disk access tasks and parallel programs (at least coded to be run in parallel, without regard of scheduling). That may occur constantly or in bursts in some real-world applications. The latter two are supposed to represent more realistic overall workloads. Table 6.1 shows the direct performance comparison between the plain VM and the confidential VM for the specified benchmarks. Some interesting trends can be seen.

In the file copy benchmark, the VM incorporating SVSM performs significantly worse

than the plain VM in all scenarios. The effect, however, becomes less impactful the more confidential VMs are running. As has been stated, this test accesses the disk frequently. But, it does so using the guest's file system and also copies a lot of data to and from memory. So the issue in this case could have a few sources. The downwards trend for increasing VM count could be attributed to how the same HV orchestrates all the VMs and certain overheads could shrink when utilizing already instantiated resources. Also, the tests never reach the host's limit of available memory.

The process creation benchmark, that forks a lot of processes and kills the child processes immediately after, shows the exact opposite trend. The more VMs are started the worse the confidential VM performs in comparison to the plain VMs. For only a single VM, the confidential VM even outperforms the plain VM. Still, competition for the CPU when creating many processes in the guest seems to gravely impact confidential VMs disproportionally more than the plain VMs.

The two shell script benchmarks arguably run the most realistic workloads (compared to the previous ones). The plain VMs and confidential VMs seem to be equal in their performance until the compute power of the host is exhausted. This is likely to be the case, because in the eight VM scenarios, the host has no dedicated cores left to perform its own tasks (eight VMs times eight cores fully exhaust all 64 physical processing cores). Apparently, the plain VM has more expensive context switches in this corner case.

Table 6.1.: Comparison between the plain VM and the confidential VM when running eight processes per VM. A negative number indicates the confidential VM performs that much worse than in the equivalent case on the plain VM. A positive number indicates the confidential VM performs that much better.

Benchmark Name	One VM	Four VMs	Eight VMs
File Copy, 4096B, 8k maxblocks	-31.4%	-23.3%	-15.0%
Process Creation	+14.2%	-1.9%	-8.6%
Shell Scripts (1 concurrent)	-0.2%	-0.6%	+13.9%
Shell Scripts (8 concurrent)	-0.6%	-0.3%	+12.6%

For better understanding of the data, Figure 6.1 and Figure 6.2 visualizes the comparisons not only in respect to the different VM types, but different VM concurrency as well.

6.3.2. CVM eval

The CVM benchmark suite is dedicated to benchmarking mechanisms that relate to Qemu VMs with the aim of analyzing events regarding SEV-SNP. A major tool to enable this kind of observability is the use of eBPF programs (equivalently also called BPF programs). They are a relatively modern way of extending the kernel. The base idea is to set up a special in-kernel Virtual machine to execute bytecode of a user-defined program (this bytecode has special syntax and needs specialized compilers or other

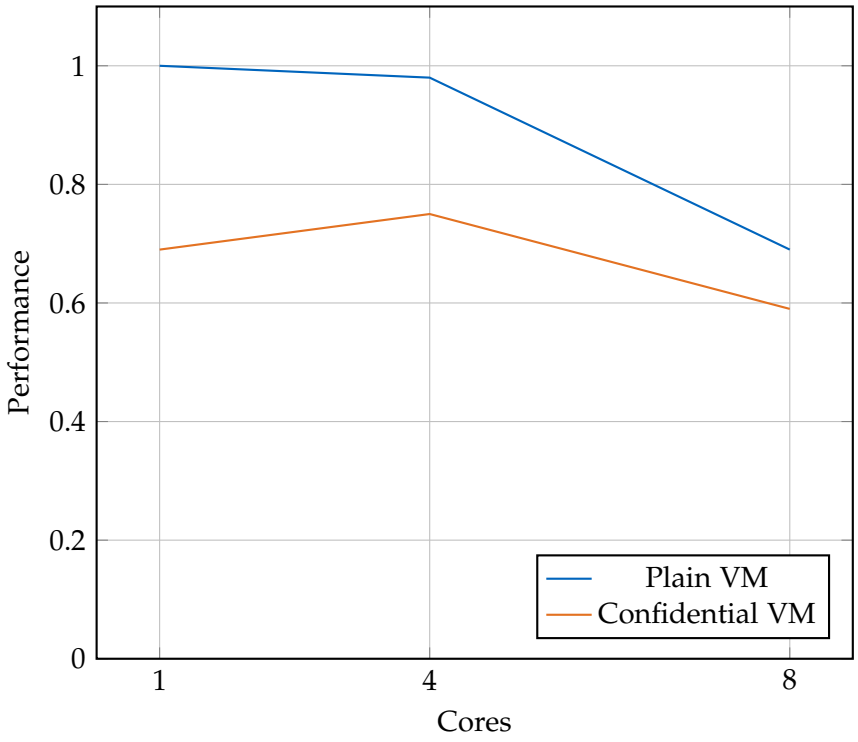

Figure 6.1.: Comparison of the file copy benchmark. The base value is the plain VM's at a single executed VM (=1). The other values are normalized to it, while a higher value means better performance.

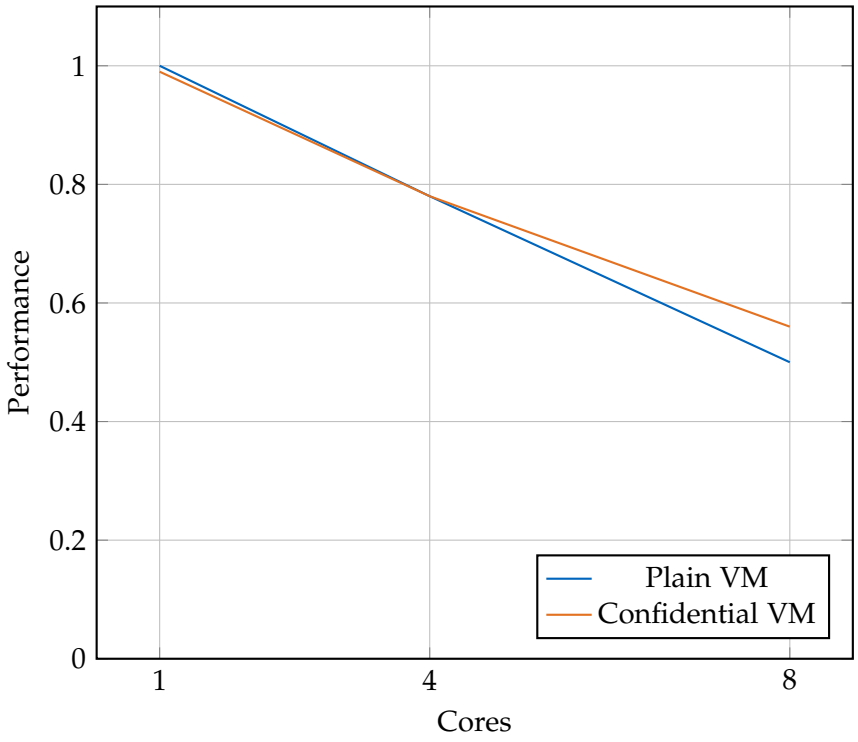

Figure 6.2.: Comparison of the shell script execution benchmark (8 concurrent scripts). The base value is the plain VM's at a single executed VM (=1). The other values are normalized to it, while a higher value means better performance.

tools to translate to it). This bytecode is verified before being loaded, thus it is ensured it cannot crash the system (which is a huge difference to kernel modules). The main idea behind these eBPF programs is, to attach them to a certain point in the kernel or user-space program. Whenever this attachment point is hit, the program is executed.

There are various eBPF program types and attachment methods, however the one that is relevant in this context is the ability to perform tracing tasks. Such an eBPF program can read all relevant information of a certain attachment point, i.e. what (kernel) function is being called with which parameters. It can then forward this information by writing to specialized and efficient data structures that can be shared with user space.

The tool that is used for the following tests is *bpftrace*[7]. It accepts eBPF programs written in a special scripting language, handles compilation, loading, attaching as well as user-space tasks like printing the tracing events. Thus, it is quite versatile and as will be shown below, can be used to easily trace virtualization related events.

As has been explained in section 6.2, it is quite crucial to have an understanding of SVSM's impact on VM boot performance. In order to measure this, timestamps of various steps in the boot process are captured, and intervals spent them can be inferred.

This uses two types of attachment points. One is a user space function. Given a binary, it can attach to its symbols with arbitrary offset. It is being used to get a timestamp of the Qemu entry point, the time `0` for the following measurements. However, the host cannot attach to trace points within the guest (especially so due to the encryption). A special trace point for KVM enables tracing events affecting a special IO port. This trace point is hit whenever the guest writes something to that predefined port. For the tests, the guest will only ever write a single byte to this port as an event to trace (as to keep additional overhead introduced by the benchmarking mechanism minimal); the bpftrace process on the host can use this value to identify which trace point has been hit.

The guest needs to be modified to emit such trace events. More specifically, the guest's firmware, SVSM kernel and Linux kernel are modified. The modified SVSM kernel emits an event on its main function entry and right before its exit. The firmware emits events for entry and exit of main functions of both PEI and DXE (UEFI stages) as well as right before launching the Linux kernel. The Linux kernel then emits a last tracing event when it has finished booting. This is determined with the help of the systemd init system. The small program causing the event to be emitted is started as a system service and executed after everything else has been initialized.

Table 6.2 shows the timings recorded in the boot process for both the plain VM and the one running SVSM. Results are averages over five boot cycles each. It can be seen that SVSM delays boot time about 3.8 seconds longer, or about 29% more time than the plain VM.

[7]Detailed info on bpftrace: `https://bpftrace.org/`

Table 6.2.: Comparison of the boot time between the plain VM and the one running SVSM. Values are given in seconds and normalized to have the entering of Qemu as time zero.

Tracepoint	Plain VM	SVSM VM
QEMU: enter main	0 s	0 s
SVSM: enter main	-	2.41432 s
SVSM: exit	-	2.47357 s
Firmware: enter PEI	0.29205 s	2.61414 s
Firmware: exit PEI	0.333627 s	3.49484 s
Firmware: enter DXE	0.339015 s	3.49927 s
Firmware: exit DXE	0.412362 s	3.6360188 s
Firmware: exit boot service	1.519856202 s	5.12994 s
Linux: systemd init end	13.0558 s	16.8548 s

7. Conclusion

The technology of AMDs confidential virtualization platform has been presented in detail. This includes hardware features as well as a specific use case, regarding a security monitor running in VMs. The potential of such a security monitor in respect to very secure deployment options of VMs on commercial and/or off-premises cloud solutions has been discussed.

Furthermore, an extension to SVSM has been proposed that increases security even more in such a deployment scenario. The goal is to prevent a malicious actor on the cloud provider's host to exchange the guest kernel that is booted into the VM. The unmodified SVSM version would not notice such an attack directly; information could be leaked from within the guest OS. A great step towards that goal has been taken, by introducing a protocol that hashes any memory page, including kernel pages, and sends a signed version of this hash to the guest OS. This can be used in future efforts to harden the kernel when booting.

The VMs using SVSM and those who don't have then been benchmarked. It is important to know for users, if the utilization of SVSM decreases performance of their workloads. On top of that, it is also crucial to understand how SVSM delays boot time of VMs. This is especially useful information for the cloud providers. It could be seen that under certain circumstances, SVSM's performance is significantly lower than that of a plain VM. However, these results need to be taken with a grain of salt, as they are deducted from highly artificial test cases. Apart from that, some very unexpected and hard to explain results have been retrieved. The most useful test case is likely to be the one of shell script performance. It showed insignificance performance differences for scenarios that don't fully exhaust host resources.

Boot time is significantly delayed however; almost a 30% increase may be a problem in highly flexible, elastic systems that run on huge server farms. A lot of VMs might need to be constantly started, shut down and migrated. Thus, this could result in some performance losses on the side of the provider.

A. UnixBench Results

The raw results of the UnixBench tests for different scenarios (as described per table). The concrete values are rather arbitrary; their worth lies in comparing them across scenarios. UnixBench provides baseline values in order to normalize the values when a single machine is benchmarked. Since this work has run several scenarios, this is not necessary however; values can be directly compared with each other. Empty fields (denoted with a hyphon) signify that the corresponding benchmark has not been run for this scenario. The system benchmarks have been run for all scenarios, the rest only for the case with one concurrently running VM.

Table A.1.: Detailed results of the UnixBench test suite for the plain VM and one concurrent process running the tests per VM

Benchmark Name	One VM	Four VMs	Eight VMs
Execl Throughput	480.3 lps	528.9 lps	526.4 lps
File Copy, 1024B buffer, 2k maxblocks	7087.8 KBps	7417.5 KBps	7443.9 KBps
File Copy, 256B buffer, 500 maxblocks	1760.4 KBps	1841.6 KBps	1869.6 KBps
File Copy, 4096B, 8k maxblocks	32532.9 KBps	34306.8 KBps	34478.2 KBps
Pipe Throughput	1376624.7 lps	1389852.7 lps	1376979.7 lps
Pipe-based Context Switching	35026.1 lps	35059.0 lps	36596.3 lps
Process Creation	5171.9 lps	5402.3 lps	5778.5 lps
Shell Scripts (1 concurrent)	1218.1 lpm	1139.6 lpm	976.3 lpm
Shell Scripts (8 concurrent)	328.7 lpm	266.3 lpm	173.8 lpm
System Call Overhead	1504162.9 lps	1505029.3 lps	1504876.2 lps
Arithmetic (double)	16544374.9 lps	-	-
Arithmetic (int)	16544077.7 lps	-	-
Arithoh	1974084423.5 lps	-	-
Exec System Call Overhead	1534.6 lps	-	-
Grep large file	8818.5 lpm	-	-
Recursion (Tower of Hanoi)	323591.7 lps	-	-

Table A.2.: Detailed results of the UnixBench test suite for the VM booting SVSM and one concurrent process running the tests per VM

Benchmark Name	One VM	Four VMs	Eight VMs
Execl Throughput	508.5 lps	563.2 lps	536.1 lps
File Copy 1024 bufsize 2000 maxblocks	7538.9 KBps	8169.2 KBps	7941.1 KBps
File Copy 256 bufsize 500 maxblocks	1922.3 KBps	2079.3 KBps	1949.0 KBps
File Copy 4096 bufsize 8000 maxblocks	29308.7 KBps	31185.3 KBps	30191.2 KBps
Pipe Throughput	1330554.6 lps	1333070.7 lps	1332122.9 lps
Pipe-based Context Switching	22656.3 lps	23223.4 lps	23056.0 lps
Process Creation	3789.7 lps	4031.7 lps	4004.8 lps
Shell Scripts (1 concurrent)	1209.4 lpm	1146.1 lpm	1034.8 lpm
Shell Scripts (8 concurrent)	331.8 lpm	265.3 lpm	194.7 lpm
System Call Overhead	1286573.2 lps	1293959.8 lps	1295762.9 lps
Arithmetic (double)	16502339.6 lps	-	-
Arithmetic (int)	16503565.2 lps	-	-
Arithoh	1967978614.9 lps	-	-
Exec System Call Overhead	1138.7 lps	-	-
Grep large file	8331.6 lpm	-	-
Recursion (Tower of Hanoi)	325779.5 lps	-	-

Table A.3.: Detailed results of the UnixBench test suite for the plain VM and eight ($\widehat{=}$ number of virtual processors per VM) concurrent processes running tests per VM

Benchmark Name	One VM	Four VMs	Eight VMs
Execl Throughput	4199.0 lps	3317.2 lps	2132.5 lps
File Copy 1024 bufsize 2000 maxblocks	33905.0 KBps	32671.6 KBps	24269.8 KBps
File Copy 256 bufsize 500 maxblocks	8523.1 KBps	8316.4 KBps	5958.0 KBps
File Copy 4096 bufsize 8000 maxblocks	156679.5 KBps	153590.3 KBps	108886.6 KBps
Pipe Throughput	10980571.2 lps	11100234.0 lps	10888203.0 lps
Pipe-based Context Switching	929760.5 lps	931538.5 lps	917280.5 lps
Process Creation	33351.4 lps	33034.3 lps	32716.5 lps
Shell Scripts (1 concurrent)	2757.7 lpm	2136.0 lpm	1338.8 lpm
Shell Scripts (8 concurrent)	336.6 lpm	263.2 lpm	168.3 lpm
System Call Overhead	12041853.0 lps	12035902.7 lps	11926208.8 lps
Arithmetic (double)	132410042.4 lps	-	-
Arithmetic (int)	132409355.5 lps	-	-
Arithoh	15790622492.1 lps	-	-
Exec System Call Overhead	9923.2 lps	-	-
Grep large file	81999.7 lpm	-	-
Recursion (Tower of Hanoi)	2608885.9 lps	-	-

Table A.4.: Detailed results of the UnixBench test suite for the VM booting SVSM and eight ($\widehat{=}$ number of virtual processors per VM) concurrent processes running tests per VM

Benchmark Name	One VM	Four VMs	Eight VMs
Execl Throughput	4174.8 lps	3379.2 lps	2455.2 lps
File Copy 1024 bufsize 2000 maxblocks	29107.2 KBps	31088.2 KBps	23584.9 KBps
File Copy 256 bufsize 500 maxblocks	7094.8 KBps	7709.0 KBps	5771.2 KBps
File Copy 4096 bufsize 8000 maxblocks	107476.1 KBps	117848.7 KBps	92533.9 KBps
Pipe Throughput	10299030.6 lps	10620014.8 lps	10620021.7 lps
Pipe-based Context Switching	1083716.9 lps	1080338.0 lps	1082774.3 lps
Process Creation	38075.4 lps	32408.4 lps	29898.3 lps
Shell Scripts (1 concurrent)	2751.5 lpm	2122.8 lpm	1525.3 lpm
Shell Scripts (8 concurrent)	334.7 lpm	262.3 lpm	189.5 lpm
System Call Overhead	10325611.3 lps	10327699.8 lps	10331259.8 lps
Arithmetic (double)	132067570.3 lps	-	-
Arithmetic (int)	132067257.8 lps	-	-
Arithoh	15746501051.4 lps	-	-
Exec System Call Overhead	10224.6 lps	-	-
Grep large file	60219.7 lpm	-	-
Recursion (Tower of Hanoi)	2606941.5 lps	-	-

List of Figures

List of Tables

Bibliography

[1] C. Bilbao. "The Linux SVSM Project." In: *lwn.net* (Jan. 2023).

[2] S. Chourasiya, G. Samanta, D. K. Sardar, P. Sharma, and C. V. Kumar. "Pegasus Spyware: A Vulnerable Behaviour-based Attack System." In: *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*. 2023, pp. 287–292. DOI: 10.1109/ICECAA58104.2023.10212163.

[3] *COCONUT-SVSM Documentation*. Last accessed 10 July 2024.

[4] D. Faggioli. *QEMU & KVM Automated Performance Benchmarking SUSE*. SUSE Labs Conference 2021. Last accessed 3 November 2024. 2021.

[5] D. Goodin. *What we know about the xz Utils backdoor that almost infected the world*. Article on Arstechnica. Last accessed 13 July 2024. Apr. 2024.

[6] *Intel Software Guard Extensions Programming Interface*. Reference # 329298-002. Intel. Oct. 2014.

[7] D. Kaplan. *Protecting VM Register State with SEV-ES*. AMD White Paper. Feb. 2017.

[8] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. AMD White Paper. Oct. 2021.

[9] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre attacks: exploiting speculative execution." In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742.

[10] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang. "A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP." In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 337–351. DOI: 10.1109/SP46214.2022.9833768.

[11] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde. *WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP*. 2024. arXiv: 2404.03526 [cs.CR].

[12] *Secure Encrypted Virtualization API*. 0.24, Revision 3.24. Publication #55766. AMD. Apr. 2020.

[13] *Secure VM Service Module for SEV-SNP Guests*. Revision 1.00. Publication #58019. AMD. July 2023.

[14] *SEV Secure Nested Paging Firmware ABI Specification*. Revision 1.55. Publication #56860. AMD. Sept. 2023.

[15] *SEV-ES Guest-Hypervisor Communication Block Standardization*. Revision 2.03. Publication #56421. AMD. July 2023.

[16] *Strengthening VM isolation with integrity protection and more*. AMD White Paper. Jan. 2020.

[17] The kernel development community. *Page Tables*. Last accessed 10 July 2024.

[18] *Trusted Platform Module Library Part 1: Architecture*. Family "2.0", Level 00 Revision 01.83. TCG. Jan. 2024.