# Transcending the Teetering Tower of Trust

## Demonstrated with Virtual Memory Fuses for Software Enclaves

Scott Brookes
Draper Laboratory
Cambridge, MA, USA
sbrookes@draper.com

## ABSTRACT

When it comes to security, there is a dangerous disconnect between the mental model we use for the modern computation stack and the real thing. The model we all use is not rich enough to understand the constant battle between attackers and security researchers across the layers of the stack. This paper offers an enhanced model and explains how security fits into the picture. We use our model to explain the implications of various types of defensive mechanisms. Then, we practice the recommended method for future security research by designing an operating system feature – software-enforced trusted execution enclaves using Virtual Memory Fuses (VMFs) – with the best-practices we argue for. The Teetering Tower of Trust model offers a new way to think about security across the computation stack, while the novel Virtual Memory Fuse creates the possibility of a new operating system feature: software enclaves.

## CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

**ACM Reference Format:**
Scott Brookes. 2020. Transcending the Teetering Tower of Trust: Demonstrated with Virtual Memory Fuses for Software Enclaves. In *New Security Paradigms Workshop 2020 (NSPW '20), October 26–29, 2020, Online, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3442167.3442168

## 1 INTRODUCTION

Most application-level security-conscious developers have probably heard (or said) something like: "if the operating system is compromised, we've already lost." Most system security researchers have probably heard (or said) something like: "if the hardware is compromised, we've already lost." But, are these statements true? Do they *need to be* true? In this paper, we aim to understand why these statements are true right now and what the security community, and operating system designers specifically, can do about it.

In Section 2, we examine the conventional model of the computation stack that lies underneath statements like these. The model is adjusted to more closely resemble the real state of the world: from a "stack" to a "Teetering Tower of Trust." With a more realistic model, we can describe the security landscape across all layers of the computation stack.

With a working mental model of not just the computation stack, but also the way security interacts across the layers, Section 3 characterizes the common approaches to security. It identifies three different methods in terms of the impact each has on the tower of trust; one that may be hurting more than helping, and two which future security efforts should strive for.

Section 4 is a design exercise exploring how operating systems designers can put the lessons learned in Section 3 into practice. After some technical background in Section 4.1 and a statement of assumptions in Section 4.2, we will explore the security posture and implementation of a novel defensive mechanism: *software enclaves*. In order to offer stronger security properties to applications, Section 4.3 introduces a new security primitive for system software: the Virtual Memory Fuse (VMF). With this tool, Section 4.4 presents the skeleton of an operating system implementation of software-enforced secure execution enclaves. In a software enclave, a sensitive process can maintain confidentiality and integrity of its code and data even in the event of a complete kernel compromise.

Section 5 offers discussions of the teetering tower of trust model and the software enclave design exercise, including thoughts about the techniques' applicability to hypervisors.

Finally, Section 6 offers concluding thoughts and main takeaways.

This work makes the following contributions:

(1) Describe a model for the current computation stack and its implications on security across its layers. (Section 2)
(2) Characterize the effects of different methodologies used by defensive security mechanisms and offer the methods most beneficial for future efforts. (Section 3)
(3) Present an operating system feature: software enclaves. Explore the implementation of this feature and how it conforms to the methodologies for transcending the Teetering Tower of Trust. (Section 4)

## 2 THE TEETERING TOWER OF TRUST

All computer scientists have seen a figure like Figure 1. Whether it was in a systems or security course or in so many systems security research papers, this figure has defined the mental model many use to think about the modern computation stack. However, this figure does not make any statement about security. In fact, many security-conscious computer scientists may be hard pressed to define the mental model they use for understanding security in the computation stack.

This traditional computational stack diagram is not so far from one that can be used to model a top-level security story for the


*NSPW '20, October 26–29, 2020, Online, USA*

ACM ISBN 978-1-4503-8995-2/20/10…$15.00
https://doi.org/10.1145/3442167.3442168

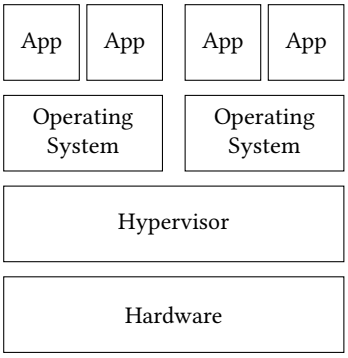


**Figure 1: The conventional computation stack. This is the model most computer scientists use to understand the layers of software in the modern computation stack. It has applications at the top of the stack with more privileged software layers underneath. Software rests on top of hardware at the base.**

modern computational landscape. With just a few changes adding features relevant to security, the figure can be the basis for a principled way to classify and evaluate broad swaths of defensive cybersecurity mechanisms across the layers of computation. In particular, there are two changes necessary to produce a figure we can use to tell a security story: the role of hardware and the relative size of the components.

Having "hardware" at the bottom of the stack is misleading. The hypervisor is not the only software component sitting on the hardware; hardware implements *all* computation throughout the entire stack. This is how Rowhammer can cause user-space bitflips [17] and how Spectre [20] can cause false branches to be executed, all from user-space. In other words, all software is realized by hardware. Using the tower analogy, it is as if hardware forms the bricks that each level is built from.

In another sense, some hardware does belong at the bottom of the stack. After all, user-space does not interact with hardware abstractions in the same way that the operating system or hypervisor do. To express this distinction, we put *hardware abstractions* at the bottom of the stack. These are specific hardware features such as the interrupt controller or paging. Abstractions like these are hidden from higher layers of the stack.

In Figure 1 each layer is drawn smaller than the layer below it. This is mostly for convenience' sake: there are more applications than operating systems, so they are drawn such that they all fit nicely on top of the OS. However, this reinforces (or is a product of) a security story that we want to believe: that good security practice demands that the lower layers are more powerful than the upper layers. In reality, the higher layers almost universally dwarf the lower layers in ways that matter for security such as interface surface, complexity, and lines of code.

Figure 2 makes these adjustments to Figure 1. This figure illustrates the title of the paper: "The Teetering Tower of Trust." Each layer is larger than the one below it. While the bottom-most layer is hardware abstractions, remember that the blocks shown at each layer are realized by hardware.

Another implication of the tower of trust on our security model: the *surface area* of one layer on another is a meaningful analogy[1]. With a larger surface area between a higher and lower layer, it is more likely that even a small compromise in the lower layer will disrupt the higher one. For instance, consider Meltdown [23]. Operating systems that use both paging and access control bits to implement their isolation schemes (higher surface area of applications on the OS) are vulnerable to the attack, while systems such as KPTI [26] or KUCS [7, 10] that use only paging (lower surface area) are not.

The tower of trust reveals that each higher layer is completely dependent on the security of *all* layers below it [1]. The irony here is that our secrets – our bank accounts, communications; all of our data – live in the highest layer, which has not only the largest attack surface on its own, but is the most vulnerable to attacks elsewhere in the stack! The deeper irony is that the intermediate layers exist purely to support and protect the highest layer; yet they have become one of the most dangerous parts of the application's footprint.

[1]This is not to say that relative surface areas in Figure 2 are meaningful. The figure is not drawn to any scale and is simply for illustrative purposes. Similarly, surface area between different userspace applications is not meaningful.

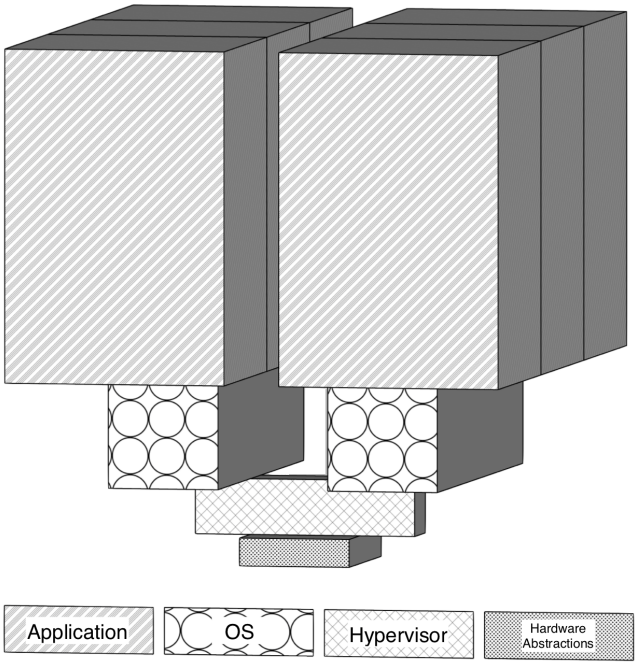


**Figure 2: The Teetering Tower of Trust. Unlike the conventional computation stack model, the higher layers of the stack are larger than the lower layers. "Hardware" is not on the bottom because hardware actually realizes every layer of the computation stack. Instead, hardware abstractions lie at the bottom, as these are the interfaces used by the lower level software.**

## 3 TRANSCENDING THE TOWER

We can categorize defensive efforts in terms of the tower of trust as a model for our computation stack and its overall security. There are three main approaches to security in terms of the model:

(1) Use some lower layer (usually $n-1$) to harden layer $n$
(2) Harden layer $n$ on its own
(3) Reduce the extent to which layer $n$ depends on layer $n-1$

*Turtles all the way Down.* Using some lower layer of the stack (typically but not always layer $n-1$) to offer some feature to harden layer $n$ leads to a natural question: "how can we protect $n-1$, then?" Without changing the approach, the answer amounts to "it's turtles all the way down!"[2]

However, many of the most common defensive techniques use this "turtles all the way down" approach. Hypervisors offer a clear example. Although they are not inherently subject to this approach, the common case for hypervisor research has made this method the most popular way to add security for the operating system. Dozens of security projects put security monitors into the hypervisor. A few examples include [28, 31, 33].

Unfortunately, this method only reinforces the issues with the tower of trust model. First of all, it almost always involves *adding* code and complexity rather than removing it. As these are directly related to insecurity in general, and for the operating system in particular [4], this is a step in the wrong direction. Moreover, the tendency to use hypervisors to push conventional security paradigms one step deeper in the stack is not changing the game for attackers, just moving it; remember, it's turtles all the way down. This is not an effective use of the powerful abstractions offered by the virtualization features of modern hardware [6, 32].

Additionally, the mechanisms using this technique encourage the attacker to target lower layers of the stack. Not only does a technique following this method make the higher layer more challenging to attack, but the lower layer also becomes more *valuable* to an attacker. This accelerates the rate at which the attacker's increase in effort to attack $n-1$ (rather than $n$) is overcome by the increasing value in attacking the lower layer. This is bad news for defensive security, which is much better prepared for dealing with attacks from above than those from below.

*Self-Hardening.* Hardening layer $n$ on its own offers a better security posture in terms of the tower of trust. When using this approach, a layer tries to make it more difficult for attackers to compromise its security properties *without* any help from lower layers. This method is most commonly applied at the operating system level, where there are many examples of security mechanisms not relying on a more privileged hypervisor actor. Some examples include the seL4 verified microkernel [19], returnless kernels [22] and Linux' Self Protection Project [15].

It is important to note that not all "hypervisor" applications use the previous method of security. One such example is ExOShim [8], which enforces execute-only memory protection on the operating system code using Intel's Extended Page Tables (EPT). Although these are hypervisor features, ExOShim doesn't exactly load a hypervisor. Instead, the kernel enables these features and then configures the ring -1 layer such that no code can ever run there again. It ensures that the protection offered by ExOShim cannot be disabled, hijacked, or modified even in the event of complete kernel compromise. This is a property the kernel has configured and enforced on itself, and the property is immutable for the lifetime of the system.

While self-hardening can be more difficult for application code because it has less freedom over the system's configuration, it is not impossible. In fact, some of the strongest techniques for security available in userspace today are examples of this method, including using memory-safe languages such as Rust and using formal verification to prove code correctness.

Unlike the first method, self-hardening does not increase the value to an attacker of compromising the lower layer. However, as it is still decreasing the difference in difficulty of targeting the lower layer versus the higher, it still pushes the attacker towards targeting lower layers of the stack.

*Comparing Turtles all the way Down and Self-Hardening.* We will use software diversification [21] (specifically ASLR) as a single mechanism that can be used to compare and contrast these two methods of security. Traditional Address-space Layout Randomization (ASLR) relies on the kernel to randomize the layout in memory of the user process at load-time. This makes it more challenging for an attacker to locate a known vulnerability, but it relies on the operating system's security. An information leak in the OS can allow an attacker to completely bypass the randomization [35].

Alternatively, ASLR at the kernel level (KASLR) [14] is a self-hardening approach that typically does not rely on the hypervisor for its security. The kernel uses its greater level of control over the system layout to randomize *itself* at its own load-time. As it is the only actor involved in the randomization, it is the only actor that can betray its randomization via an information leak. While a compromised hypervisor could read its memory, it would still require non-trivial reversing to discover the layout that the kernel is using.

This is not to say that self-hardening versus hardening by lower layers is as simple as kernel versus userspace. Compile- [21] or development-time [2] diversification can randomize an application without relying on the operating system. These techniques are similar to KASLR: a compromised operating system could read the process' memory, but it would need to reverse engineer the memory to determine how the process is laid out. These techniques harden the application layer without the support of the operating system, therefore they offer a stronger security posture than classic ASLR.

*Self-Reliance.* The most effective method for security in terms of the tower of trust is to stabilize the tower by bypassing intermediate layers. Although we must always have some amount of trust [37], if the application layer rests directly on hardware abstractions it is no longer vulnerable to compromise in the operating system or hypervisor layers. These types of mechanisms not only increase the difficulty of attacking a higher layer, but they also decrease the incentive for the attacker to target intermediate layers because they do not immediately achieve compromise of the higher layers.

Although not as much work has taken this challenging approach, there are some examples, including Unikernels [5]. By running just

[2] "It's turtles all the way down" is a way to describe a "chicken or egg" style recursive problem. While its origin is unknown, it is most commonly described as an old woman telling a scientist that the world is flat and carried on the back of a giant turtle, standing on a larger turtle. When asked what the larger turtle stands on, she says "it's turtles all the way down!" [38].

a single process per operating system and using a hypervisor to handle multitasking, the attacker is disincentivised to compromise the operating system layer and forced to either attack the application or the hypervisor directly.

Taking the protected process one step further, hardware Trusted Execution Enclaves (TEEs) such as ARM's Trustzone [29] or Intel's SGX [13] place a secure process' trust in a hardware feature rather than in the operating system. This means that compromising the operating system does not compromise the process. The attacker is forced to choose between attacking the hardened protected process directly, or attacking the hardware itself.

*Characterizing Security.* Turtles all the way down, self-hardening, and self-reliance are underlying methods for a *single mechanism* to deliver security rather than possible classifications for the security posture of a system.

Trying to use the three methods of security to describe the security posture of an entire system will always bottom out on turtles all the way down because all software must always trust some hardware. Insofar as they are used to describe the underlying method of a single security mechanism, they describe the effect that the isolated mechanism has on the overall security posture of the system.

For example, a system using Unikernels is not "secure by self-reliance" because it still relies on the security of the hypervisor and hardware. However, deploying Unikernels is using self-reliance to harden the entire system by removing the operating system from the trusted computing base.

Similarly, a formally verified system still trusts the integrity of $n - 1$ layers. However, it has opted to make the system harder to attack by deploying defenses in its own layer. Thus, formal verification as a mechanism is self-hardening.

# 4 DESIGN EXERCISE: SOFTWARE ENCLAVES

The rest of this paper presents a design exercise intended to explore a security mechanism that embraces self-reliance: decreasing the higher layer's dependence on a lower layer. In particular, we will explain how operating systems could leverage the simplest Intel x86 hardware abstractions to offer process sandboxes that the operating system itself cannot read, write, or execute. These have similar properties to a TEE hardware enclave, but they are more flexible and do not rely on sensitive hardware implementations; they are effectively *software enclaves*. This operating system feature will not only show how applications need not rely on the operating system, but also how software can minimize its hardware footprint as insurance against some hardware failures.

First, Section 4.1 gives a brief background on operating systems, virtual memory, and enclaves. We then scope the exercise with some assumptions in Section 4.2. In Section 4.3, we begin the exercise with a novel software architecture primitive: the Virtual Memory Fuse (VMF). Section 4.4 discusses an operating system design that uses VMFs to implement secure software enclaves.

## 4.1 Background

*4.1.1 Virtual Memory*[3]. The virtual memory abstraction is a fundamental hardware feature of modern computational systems. With virtual memory, software uses addresses that do not correspond directly to physical addresses in memory. Instead, the hardware translates virtual addresses used by software through a configurable set of tables in order to produce a corresponding physical address.

On a particular x86 processor core, Control Register 3 (`CR3`) defines the virtual memory context of the core. This control register contains the physical address of a base level paging structure. From that address, the memory management unit follows a chain of physical address pointers down through the multi-level paging structures until it resolves the physical address corresponding to the inputted virtual address as a function of the currently loaded paging structures.

*4.1.2 Operating Systems.* It is difficult to define the term "operating system." For the purposes of this paper, it is sufficient to describe the operating system as that software which:

- Runs in Ring 0 on an Intel processor
- Loads, schedules, and serves Ring 3 processes
- Multiplexes and manages access to hardware resources, including memory

Different operating system architectures offer different security and performance tradeoffs. Traditional monolithic operating systems are the most functional and fastest, while microkernels are more secure but come with more performance issues. Unikernels [5] are stripped-down kernels running just a single process. Any of these architectures could implement software enclaves as discussed below, each with its own unique challenges.

*4.1.3 Enclaves.* An enclave is a form of Trusted Execution Environment (TEE). TEEs offer a hardware-enforced[4] boundary for software to be protected outside of the typical privilege mode paradigm. The most common examples are ARM's TrustZone [29] and Intel's SGX [13]. These and other TEE technologies are surveyed in [39, 40].

While an enclave is most commonly used as a protected space for an application to execute (employing "self reliance"), it can also house software that provides integrity to the rest of the system (a turtles all the way down approach). Two examples of the latter approach are SPROBES [16] and TZ-RKP [3]. Both of these techniques force key memory manipulations to take place in the enclave, pulling trust out of the operating system and locating it in a more privileged layer.

It is important to note that attackers have compromised both SGX [11, 27] and TrustZone [36] using hardware-based attacks. Furthermore, all enclaves (including software enclaves) are vulnerable to Iago attacks [12] in which a corrupted operating system returns malicious system call responses to the unsuspecting enclave.

The intent in this section is to design a software-defined enclave that offers similar security properties to TrustZone and SGX.

## 4.2 Assumptions and Threat Model

Our mechanism's goal is to protect the confidentiality and integrity of a protected process' code and data, even if another malicious process has escalated privilege to that of the OS. As such, we assume

[3] Subsection text from author's prior work [7]

[4] The only other "software-based" enclave technology we found was SecTEE [41] which still requires ARM TrustZone: stating that "SecTEE is designed to be incorporated into the TrustZone software architecture."

that after a protected process has been launched, a remote or local attacker manages to escalate privilege [9] to that of the operating system. The attacker is attempting to extract or control some data stored within the boundaries of the process. The attacker may execute arbitrary code with kernel privilege and read or write all memory in the kernel's address space. The attacker does not have any information from before the initialization of the protected process.

We will target the x86-64 architecture. We assume that the operating system already has the process running in a separate address space from itself e.g. Linux with KPTI [26] or Kernel and User Core-based Separation (KUCS) [7, 10]. The design assumes a secure boot and initialization time; attacks before the start of the protected process are left as future work. The availability property of the trusted process is out of scope; a compromised OS may deny service.

While this work is motivated to design software that can survive certain hardware compromises, we do not claim to survive arbitrary hardware failures. We assume that the core virtual memory abstraction is intact. In particular, we assume that all memory accesses are interpreted as virtual addresses by the MMU, and correctly resolved by the page tables defined by the `CR3` register. We assume that a physical address not present in those tables cannot be accessed. We will not assume the correctness of the permission bit in the tables.

The design exercise develops software enclaves to showcase the self-reliance method of security. Importantly, we are not seeking to solve the larger problems inherent to secure enclaves in general.

Finally, side-channel attacks are out of scope. While providing stronger separation between user and kernelspace may make some side-channel attacks more difficult, we do not claim to defeat all side-channels. Side-channel attacks deserve their own analysis which is beyond the scope of this paper.

## 4.3 The Virtual Memory Fuse

In order to implement our software enclaves, we will first introduce a new system software architecture primitive: the Virtual Memory Fuse (VMF). Broadly defined, the virtual memory fuse is the set of conditions that enable one operating context to read, write, or execute the memory of another. We call these conditions a "fuse" because invalidating the conditions (i.e. "blowing the fuse") results in two completely isolated contexts.

In order for operating context $S$ (subject) to access (i.e. read, write, or execute) a region of memory $O$ (object), $S$ must possess a virtual mapping to $O$. This is the single most fundamental property of the virtual memory paging abstraction discussed in Section 4.1.

However, this single condition is not sufficient. In order to guarantee that $S$ cannot access $O$ it must have neither a mapping to $O$ *nor* the ability to create such a mapping. Consequently, the Virtual Memory Fuse from a subject to an object is made of two parts:

(1) A valid virtual mapping to the object, and:
(2) The ability to make a virtual mapping to the object.

A trivial example of the VMF in practice is userspace's inability to access kernelspace memory. In the traditional kernel/userspace split between ring 0 and ring 3 on an x86 architecture, mappings for the userspace memory are set up with a lower privilege than the mappings to the kernelspace memory, including the address space's page tables. This denies both conditions of the VMF, effectively "blowing" the fuse in order to protect the operating system from user processes.

## 4.4 Implementing Software Enclaves

By blowing the Virtual Memory Fuse, the kernel can generate a protected enclave we call a VMF Enclave (VMFE). We will show how the kernel can create such a memory space before discussing the implementation details of two separate operating systems that might implement this feature. One is targeted for systems that have a narrow and well-defined workload with long-running processes (e.g. control systems, embedded systems) while the other begins to tackle the additional challenges of a server-class system's unpredictable workload.

*4.4.1 A VMF Enclave.* The VMF Enclave is an address space with a particular virtual memory configuration in which a normal userspace application can be loaded and run. The VMFE ensures that the operating system cannot read, write, or execute the process' memory, even if it is completely compromised.

To launch a VMFE the operating system, at a high level, needs to relinquish some of its control over the system in order to create a memory space that even it cannot introspect. The process runs in this space, safe from the prying eyes of a kernel subjected to a confused deputy [18] or other type or privilege escalation attack [9].

It is nontrivial to blow the fuse from kernel to userspace because the operating system is more privileged than the process. As such, it has permission to access all of the memory on the system and therefore both parts of the VMF are intact.

Blowing the first part of the fuse – active mappings to the object – is quite simple. The kernel can modify its own page tables to remove mappings to the VMFE. Although most kernels do rely on having the process mapped into their address space with the same layout used by the process itself, [7, 10] have discussed at length how to implement an operating system that does not require these mappings. That work did maintain mappings to the process, but they were ad-hoc (i.e. they did not preserve the layout of the process).

Ostensibly, blowing the second part is equally simple. The kernel will delete the mappings to its own page tables so that it cannot re-map the VMFE memory. Because all memory accesses operate through the MMU, even page table memory needs corresponding page table entries in order to be written to. Therefore, stopping the kernel from writing to the page table is as simple as stopping the kernel from writing to the VMFE itself. Figure 3 shows the virtual memory layout of a VMFE loaded on an operating system using KPTI [26] style virtual memory management.

*The remaining attack vector.* Unfortunately, deleting the page table mappings does not stop the kernel from creating an entirely new page table with the necessary mappings and switching to it as the active context. Although this would be very difficult for an attacker, switching to a new context with a carefully crafted new page table structure could compromise the VMFE. Part of the challenge of performing this attack has to do with the careful saving and restoration of state associated with a context switch. In particular, the program counter (`RIP` on x86-64) *does not change* during a context switch. This means that the entry point for new

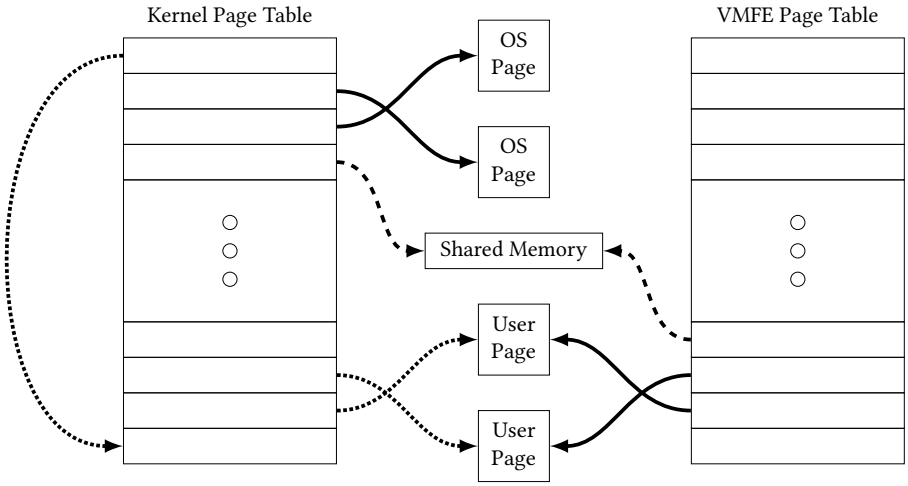


**Figure 3: Memory layout for a VMFE with a KPTI style operating system. The dashed arrows are new mappings to a shared memory region that the process and kernel use for passing information e.g. system call arguments or return values. The dotted arrows are deleted virtual mappings as part of blowing the VMF. The solid arrows are existing virtual mappings that remain throughout run-time.**

code needs to be loaded at the same address as the exit point for the old context. In particular, successfully switching contexts from one active page table to another requires that the physical address of the entry point for the new context's code must be loaded at the virtual address of the exit point of the old context's code. We present a few methods for defeating this attack:

- The kernel could delete bookkeeping/records such that it no longer knows what physical memory the protected process is using. If the process were loaded with some randomization and the kernel did not have physical addresses linked from its process structure, an attacker would not be able to construct a page table with the process at known virtual addresses.
  This method would not interrupt the kernel's behavior much at all. The only challenge would be cleaning up the process on exit. In order to simplify this task, the process itself could record its physical addresses (as reported by the kernel during initialization, before blowing the VMF) and when it exits, it could simply report its physical addresses back to the kernel for cleanup.
  Certainly, an attacker could still defeat this method. The attacker could simply map all physical memory into a page table, at which point they would be able to read the memory of the entire system. Finding the process would require non-trivial reverse engineering, though the task would be slightly more simple because the process would be somewhere in a limited amount of physical memory about which the kernel has no ownership data, amongst hardware mapped regions and other reserved physical memory "holes". While this technique does increase attacker workload, its main benefit is limited disruption to the kernel rather than actually stopping the attack.
- Most earnestly, the kernel could delete all records of virtual-to-physical mappings such that it would be too ignorant to conduct the attack. Without any such mappings, the attacker could not know the physical address of the payload code, meaning they would be unable to produce a valid target context page table. Although this would have far-reaching effects on the kernel's ability to perform memory management, process creation and deletion, device management, and more, an embedded kernel with a well defined workload may actually be able to operate uninterrupted. However, we will consider embedded as well as server-class operating systems and solving these problems for the latter case is beyond the scope of this exercise (though we do believe it is possible).
- The kernel can implement a policy in the hypervisor layer that traps on writes to the `CR3` register and verifies that only pre-approved `CR3` targets have mappings to the protected process. This method, depending on its implementation, could be a classic turtles all the way down approach *or* a self-hardening approach similar to ExOShim [8]. In the latter case, the "hypervisor" can be considered as a well-isolated bit of trusted kernel functionality.

For the remainder of the design exercise, we will assume that the hypervisor solution is employed. Although careful implementation would be required to avoid a turtles all the way down type solution, we are moving forward with this assumption to simplify the discussion and avoid broadening the scope of the paper.

*4.4.2 Software Enclaves for Embedded Systems.* As blowing the VMF for a VMFE is quite disruptive to the capabilities of a traditional operating system, we will start our exploration with an embedded operating system which may have lower requirements for long-term flexibility based on having a well-defined workload.

For an embedded system, we assume that the process(es) that need to run in a VMFE are well defined and known at boot-time.

This means that the kernel only needs to burn its VMF(s) a single time for the lifetime of the system rather than needing to constantly instantiate new VMFEs for processes that arise at run-time.

After booting and blowing the VMF, the kernel will no longer be able to modify page tables. However, it can still perform much of the work required of an operating system without interruption:

- *Serve system calls:* with the correct support at the library layer of the application, the process can communicate with the kernel through only a narrow interface of shared memory. This is discussed in [7].
- *Schedule:* the kernel can run its scheduling algorithms without any knowledge of the memory layout. Furthermore, scheduling a new process requires only the physical address of the new context's CR3 target; not even a virtual mapping to that address is needed. The kernel could schedule with no issues.
- *Process Management:* as discussed earlier in terms of defeating the VMF, the kernel can create and launch new page tables at will. This means that loading new processes will be possible. We also discussed process exit and destruction.

Of course, other tasks are more challenging. In particular, conventional operating systems have a unique OS instance per process (unlike KUCS [7]). If one of these operating systems needs to fork new processes, it must duplicate the operating system context. This normally straight-forward copying of the active page tables is a major challenge without mappings to the kernel's own page tables.

While the kernel could simply maintain information about its own layout in memory separate from its page tables, and refer to that when creating a copy of itself, this problem offers a chance to introduce the notion of a VMF Kernel Enclave (VMFKE). There is no reason that a VMFE needs to contain a userspace process. A kernel component could be loaded into a VMFE to make a VMFKE. To address this particular issue the VMFKE would not burn its VMF entirely (it would maintain mappings to its own page tables). As such, any running VMFEs would have to trust this component; keeping the code small, formal verification, and/or careful isolation of recordkeeping (i.e. don't give this component access to process memory addresses) would be required, but the main operating system still need not be trusted.

*4.4.3 Software Enclaves for Server-class Systems.* VMFEs would be very useful in a server-class system. After all, SGX was developed primarily to allow residents of the cloud protection from untrusted hosts, and a VMFE offers similar security properties. At the limit, all applications would blow their VMF and the kernel would persist to provide non-memory related services such as interfacing with and multiplexing hardware, scheduling, and handling system calls.

However, working with a server-class operating system poses additional challenges to providing VMFEs. One main question is how the operating system could recover from having blown its VMF if the VMFE were to exit and allow the OS to return to its state of full functionality.

One option is that a VMFKE could restore the kernel's control over virtual memory. Upon process exit, the VMFKE would write new mappings to the kernel's page tables. With the ability to modify its page tables, the kernel can regain all control over the system gracefully.

The next challenge will come from the long-running nature of the system. The assumption that the kernel would have time before launching the VMFE without being attacked to configure correctly is not fair in a server-class system. Some additional component would be needed to verify that the kernel is in a known-good state before launching the VMFE. A hypervisor, formal verification, or hardware (e.g. Intel TXT or custom) are all options for providing that guarantee.

# 5 DISCUSSION

## 5.1 The Teetering Tower of Trust

Much of the discussion at the workshop centered around the teetering tower model and its limitations. Several questions were raised that deserve further research, including:

- Can the model help to represent distributed systems? Perhaps as multiple towers with some sort of connection between them? Would these connections strengthen or weaken the security posture overall?
- Does this model work for other types of "stacks" in computation, e.g. the TCP/IP stack?

One conference participant noted that all metaphors have a breaking point at which they no longer manage to describe their counterpart. An especially insightful question at the workshop identified such a breaking point: its ability to capture time. How would periodic refresh fit into the model? Its not clear that this metaphor is strong enough to capture a time dimension at all. We think this deserves future research.

Finally, part of the workshop discussion showed how the model could be used to help understand security and bootstrap new ideas. A participant noted that with a real teetering tower, we would not allow people to stand around trying to topple it; we would hire guards to protect it. The ability to use the mental model to think about security mechanisms is exactly the aim of the teetering tower of trust.

## 5.2 Software Enclaves

We have described an operating system implementation that can offer software enclaves (VMFEs), giving processes the promise that they can maintain confidentiality and integrity, even in the event of complete kernel compromise.

This technique is an example of self-reliance: decreasing a layer's dependence on the layer below it for its security properties. As such, the security posture of this technique is strong. It offers strong security properties for userspace without adding much complexity to userspace and without increasing the value of compromising the hypervisor or the kernel. It even has a small footprint on hardware, relying only on the most fundamental properties of paging and virtual memory.

The design is not just a useful pedagogical exercise. In the embedded case, especially when the kernel may know all the processes it expects to run in advance, the design offers a compelling way to implement secure software enclaves. This is especially valuable in the embedded case where hardware may not have TEEs, or where overhead associated with hardware TEEs might threaten performance constraints.

*5.2.1 Hypervisors.* A similar design exercise targeting the hypervisor layer may be especially fruitful. The specific techniques developed here are applicable to hypervisors as well as operating systems. In particular, the hypervisor can blow its VMFs to its operating system guests the same way our OS did for its processes. In fact, this application is an especially good fit for these techniques because hypervisors often offer static configurations that are known at boot time [24, 25, 30], invalidating many of the concerns discussed in the sever-class OS case.

Hypervisor architectures have already explored loading their own functionality in less privileged domains from Xen's `dom0` to more advanced architectures such as Nexen (and others) as described in [34]. These distributed micro-kernel style hypervisor options are clearly more secure than a monolithic hypervisor, but they could take security to the next level by creating enclave-type protections for these modules.

Hypervisors are a good candidate for this style of defensive mechanism more broadly. As the most privileged layer of the software stack, they do not have the luxury of using turtles all the way down style approaches. They also have more powerful hardware abstractions available to implement self-protection and self-bypassing techniques (e.g. extended vs. normal page tables), but do not have access to hardware TEEs for launching guest operating systems. We believe that attackers have only scratched the surface of discovering hypervisor vulnerabilities and ideas like those discussed in this paper will be vital for securing these important software layers in the future.

## 6 CONCLUSION

In conclusion, we summarize our position as follows: trusting system software is 1) dangerous and 2) not as necessary as convention may suggest. To support this argument, we:

- Critically examined the traditional model of the "computation stack." We identified ways in which the traditional stack is not faithful to the actual state of the world and corrected these inaccuracies, producing the "Teetering Tower of Trust." The tower allowed us to incorporate a security story into the model – something that was sorely missing from the conventional stack. (Section 2)
- Classified multiple methods of security in terms of their impact on the tower of trust model. We argued that the most common method of security, using a lower layer to harden a higher layer, is a case of turtles all the way down that encourages not only higher dependence on system software, but also encourages attackers to target the lower layers of the stack.
  Alternatively, we offered two methods of security that can help avoid the challenges revealed by the tower of trust model. Methods for self-hardening offer security in spite of the lower software layers. Finally, we suggested the possibility that clever system software architectures could dramatically reduce the extent to which higher layers need trust in the lower software layers. (Section 3)
- Practiced the proposed method for security by describing an novel operating system feature: software enclaves. These isolated virtual memory contexts are made possible by the Virtual Memory Fuse. As an alternative to hardware-based Trusted Execution Enclaves, they offer similar security properties (confidentiality and integrity of a process, even from a completely compromised kernel) without the limitations or complexity inherent in a pure hardware solution. (Section 4)

We acknowledge that some system software needs to be trusted and even that changing the paradigm of trusting all system software will be a slow and difficult process. However, we believe that these methods will increase security in the long term.

Moreover, we believe there are two concrete and easily implemented contributions in this paper:

(1) Incorporating the Teetering Tower of Trust, or something like it, as a more accurate mental model of the computation stack. Overlaying security onto our highest-level model of what is happening on our machines can only lead to better security efforts.
(2) Prioritizing self-hardening techniques over turtles all the way down approaches. Although rearchitecting our system software to be "trustless" may not be realistic in the short term, using memory-safe languages, leveraging formal verification, and other self-hardening techniques are possible now. They offer a substantially stronger security posture than alternatives, and should be regarded as such.

## ACKNOWLEDGMENTS

Many thanks to the NSPW community including, but not limited to, our anonymous reviewers, our shepherds Lori Flynn and Anil Somayaji, and the workshop participants who engaged in a lively and insightful discussion of the work.

I would like to thank my colleagues Chris Casinghino, Silviu Chiricescu, John Merrill, Ryan Prince, Arun Thomas, Felipe Vilas-Boas, and Curtis Walker. This work would not be possible without their support.

I owe thanks always to Rob Denz, Martin Osterloh, and Steve Kuhn. My passion in this space is a child of their friendship, and these ideas were developed over years of musings with them.

Finally, I want to thank my fiancée Lauren Mrachko. This paper was written in the midst of the COVID-19 pandemic and her support during the various "shelter-in-place" efforts was vital to the success of this work.

## REFERENCES

[1] James P. Anderson. 1972. Computer Security Technology Planning Study.
[2] Algirdas Avizienis. 1995. The methodology of n-version programming. *Software fault tolerance* 3 (1995), 23–46.
[3] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-Time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 90–102. https://doi.org/10.1145/2660267.2660350
[4] Simon Biggs, Damon Lee, and Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-Based Designs Improve Security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 16, 7 pages. https://doi.org/10.1145/3265723.3265733
[5] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257.

[6] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. 2010. VM-Based Security Overkill: A Lament for Applied Systems Security Research. In *Proceedings of the 2010 New Security Paradigms Workshop* (Concord, Massachusetts, USA) *(NSPW '10)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/1900546.1900554

[7] Scott Brookes. 2018. *Mitigating Privilege Escalation*. Ph.D. Dissertation. Dartmouth College.

[8] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. 2016. Exoshim: Preventing memory disclosure using execute-only kernel code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*.

[9] Scott Brookes and Stephen Taylor. 2016. Containing a Confused Deputy on x86: A Survey of Privilege Escalation Mitigation Techniques. In *International Journal of Advanced Computer Science and Applications*.

[10] Scott Brookes and Stephen Taylor. 2016. Rethinking Operating System Design: Asymmetric Multiprocessing for Security and Performance. In *Proceedings of the 2016 New Security Paradigms Workshop* (Granby, Colorado, USA) *(NSPW '16)*. Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/3011883.3011886

[11] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[12] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/2451116.2451145

[13] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[14] Jake Edge. 2013. Kernel address space layout randomization.

[15] Jake Edge. 2016. State of the Kernel Self Protection Project.

[16] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. arXiv:1410.7747 [cs.CR]

[17] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2015. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *CoRR* abs/1507.06955 (2015). arXiv:1507.06955 http://arxiv.org/abs/1507.06955

[18] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 22, 4 (1988), 36–38.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

[20] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19.

[21] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, USA, 276–291. https://doi.org/10.1109/SP.2014.25

[22] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with "Return-Less" kernels. In *Proceedings of the 5th European conference on Computer systems*. 195–208.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 973–990.

[24] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. 2020. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020) (OpenAccess Series in Informatics (OASIcs), Vol. 77)*, Marko Bertogna and Federico Terraneo (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:14. https://doi.org/10.4230/OASIcs.NG-RES.2020.3

[25] M Masmano, I Ripoll, Alfons Crespo, and Metge Jean-Jacques. 2009. XtratuM: a Hypervisor for Safety Critical Embedded Systems.

[26] Lars Müller. 2018. KPTI a Mitigation Method against Meltdown. *Advanced Microkernel Operating Systems* (2018), 41.

[27] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*.

[28] Nick L. Petroni, Jr. and Michael Hicks. 2007. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 103–115.

[29] Sandro Pinto and Nuno Santos. 2019. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[30] Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. 2017. Look mum, no VM exits!(almost). *arXiv preprint arXiv:1705.06932* (2017).

[31] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Recent Advances in Intrusion Detection*, Richard Lippmann, Engin Kirda, and Ari Trachtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[32] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. 2007. Hype and Virtue. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems* (San Diego, CA) *(HOTOS'07)*. USENIX Association, USA, Article 4, 6 pages.

[33] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 335–350. https://doi.org/10.1145/1323293.1294294

[34] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Deconstructing Xen. In *NDSS*. https://doi.org/10.14722/ndss.2017.23455

[35] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*. 1–8.

[36] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[37] Thomas Wadlow. 2014. Who must you trust? *Queue* 12, 5 (2014), 30–43.

[38] Wikipedia contributors. 2020. Turtles all the way down — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Turtles_all_the_way_down Online; accessed 30-April-2020.

[39] Fengwei Zhang and Hongwei Zhang. 2016. SoK: A Study of Using Hardware-Assisted Isolated Execution Environments for Security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016* (Seoul, Republic of Korea) *(HASP 2016)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2948618.2948621

[40] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. 2019. SoK: Hardware Security Support for Trustworthy Execution. arXiv:1910.04957 [cs.CR]

[41] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. 2019. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1723–1740. https://doi.org/10.1145/3319535.3363205