Blindfold: Confidential Memory Management by Untrusted Operating System

Caihua Li
Yale University
caihua.li@yale.edu

Seung-seob Lee
Yale University
seung-seob.lee@yale.edu

Lin Zhong
Yale University
lin.zhong@yale.edu

***Abstract*—Confidential Computing (CC) has received increasing attention in recent years as a mechanism to protect user data from untrusted operating systems (OSes). Existing CC solutions hide confidential memory from the OS and/or encrypt it to achieve confidentiality. In doing so, they render OS memory optimization unusable or complicate the trusted computing base (TCB) required for optimization.**

This paper presents our results toward overcoming these limitations, synthesized in a CC design named Blindfold. Like many other CC solutions, Blindfold relies on a small trusted software component running at a higher privilege level than the kernel, called Guardian. It features three techniques that can enhance existing CC solutions. First, instead of nesting page tables, Blindfold's Guardian mediates how the OS accesses memory and handles exceptions by switching page and interrupt tables. Second, Blindfold employs a lightweight capability system to regulate the OS's semantic access to user memory, unifying case-by-case approaches in previous work. Finally, Blindfold provides carefully designed secure ABI for confidential memory management without encryption.

We report an implementation of Blindfold that works on ARMv8-A/Linux. Using Blindfold's prototype, we are able to evaluate the cost of enabling confidential memory management by the untrusted Linux kernel. We show Blindfold has a smaller runtime TCB than related systems and enjoys competitive performance. More importantly, we show that the Linux kernel, including all of its memory optimizations except memory compression, can function properly for confidential memory. This requires only about 400 lines of kernel modifications.

I. Introduction

Modern operating systems (OSes) enjoy unfettered access to the application data. This access is problematic because the OS may not be trustworthy, due to vulnerabilities from its large attack surface [8] or lack of trust in the OS provider. In recent years, many have attempted to ameliorate this problem under the umbrella of Confidential Computing (CC). We say that a process or application is sensitive if it does not trust the OS; we call the memory used by such applications Confidential Memory. Existing CC solutions do not adequately support legitimate OS access to Confidential Memory and, as a result, poorly support modern big-data applications. (*i*) Some, e.g., TrustShadow [4] and BlackBox [7], hide memory used by sensitive applications from the OS, and as a result, OS functions stop working for such memory. Others, e.g., Overshadow [1], resort to expensive encryption for all OS access. (*ii*) When the OS requires clear-text access to user memory, e.g., system call arguments, existing CC solutions take a case-by-case approach, leading to inflated trusted computing base (TCB) and extra data copy. (*iii*) Many of them resort to using an additional level of address translation managed by the TCB to decouple protection from address translation [1], [9], [5], [10], [6], [7]. As a result, important OS optimization for big-data applications, e.g., page migration [11] and hugepage [12], would no longer work. (*iv*) Other hardware-based solutions, such as Intel SGX, often suffer from hardware limitations. For example, applications built on top of Intel SGX suffer from a limited Enclave Page Cache (EPC) as most platforms have 128 MB or 256 MB of Processor Reserved Memory (PRM) [13].

This paper reports our experience of overcoming the above limitations and allowing an untrusted Linux kernel to manage confidential memory without jeopardizing its confidentiality. We present Blindfold and its implementation for ARMv8-A. Like many existing CC solutions, Blindfold employs a small trusted software called Guardian that runs at a higher privilege level than the Linux kernel. With Blindfold, we demonstrate the effectiveness of a suite of techniques that can be adopted by existing CC solutions to overcome the limitations discussed above. First, unlike previous work that deploys additional nested (or shadow) page tables and interrupt tables in TCB, Blindfold keeps them out of the TCB (Guardian) but lets Guardian determine which one to use. That is, it switches between these tables, instead of nesting them (see Figure 1). Blindfold employs this idea to mediate memory access by the OS and the DMA (§IV-A) as well as protect the control flow integrity (CFI) of the protected application in interrupts (§IV-C). This technique can be used to isolate software regardless of which privilege mode it runs in, an objective of Tyche [14]. Second, unlike previous work that supports *semantic* kernel access in a case-by-case manner, Blindfold employs a single mechanism, a lightweight capability system, to support all (see Figure 2; §IV-B). A sensitive process explicitly grants a capability to the OS when making a system call. Third, to enable *nonsemantic* kernel access to manage confidential memory avoiding high overhead, Blindfold identifies the most popular memory operations in the kernel

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240294
www.ndss-symposium.org

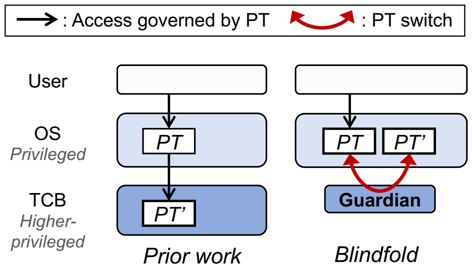

Fig. 1: A very high-level comparison of Blindfold's approach of switching vs. that of nesting used in [1], [2], [3], [4], [5], [6], [7]. In the latter, the TCB (*higher-privileged*) must manage the additional level of address translations and its page tables. In contrast, in Blindfold, all the page tables (PTs) are still in the OS (*privileged*) while the TCB (*Guardian*) only mediates their use and updates. Similarly Blindfold also places the interrupt tables in the OS while the TCB only decides which one to use depending on whether the running process is sensitive or not, unlike prior work in which the interrupt table is inside the TCB, e.g., [4], [5], [6]. See §VII for the detailed comparison.

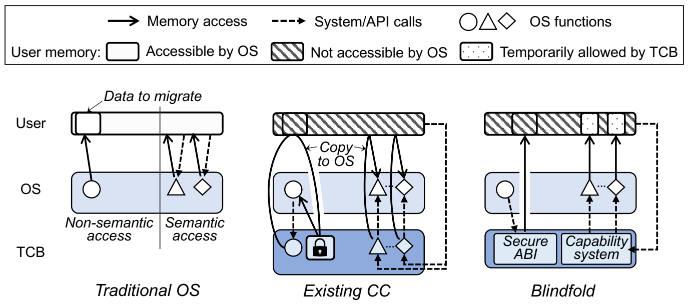

Fig. 2: A comparison of Blindfold 's techniques for *semantic* and *non-semantic* access vs. existing solutions. For non-semantic access, such as page migration, existing CC solutions either block the OS's access to user space, disabling essential OS functions, or require expensive data copy and encryption. For semantic access, such as the `write` system call, existing CC solutions employ corresponding service functions inside the TCB, i.e., case-by-case, leading to an inflated TCB. Blindfold resolves these issues by providing secure ABI and a capability system with general applicability.

and provides secure ABI for management (see Figure 2; §IV-A), instead of always providing an encrypted view of user pages to the OS as in prior work like Overshadow [1].

With Blindfold's prototype, we empirically show, for the first time, that the Linux kernel along with most (if not all) of its optimizations can indeed function properly with confidential memory. Only memory compression [15] would lose its effectiveness (but still work) under Blindfold. Importantly, we are able to quantify the cost of managing confidential memory. With proper optimization, Blindfold imposes about 3% to 25% performance overhead to unprotected, memory and compute intensive applications, while 10% to 44% to protected ones. However, it does impose substantial overhead for I/O-intensive applications, including those that require frequent restarts. Much of the overhead comes from cryptography and control-flow changes, instead of memory access. Moreover, Blindfold requires about 400 LOC of modification of the Linux kernel. Its Guardian, implemented in mostly safe Rust (2.2K LOC), is about half the size of TCB from related systems [1], [4], [7], because Guardian does not manage memory or handle exceptions.

We note that Blindfold only uses widely available and time-tested architectural supports (§III-B) and as a result it is highly portable. At the time of this writing, a basic x86-64 port (without full features) already works. Moreover, Blindfold supports legacy binaries and supports both protected and unprotected applications on the same system, with unprotected ones paying small performance overhead, except I/O intensive applications similar to BlackBox [7] and TrustShadow [4]. Blindfold is open-source and available from [16] and an early prototype of it is described in [17]

II. Background

We next provide a succinct background about OS memory management as related to Blindfold, using Linux as a concrete example.

Page Table-mediated Memory Access. Modern systems, including the kernel and I/O devices, access memory *virtually* through page table-based address translation, facilitated by the memory management unit (MMU) or IOMMU in the case of direct memory access (DMA) by I/O. The kernel can freely access the user space with the user page table: it enjoys the privilege of determining which page table to use and of changing a page table entry. The kernel can also access physical frames that host sensitive user data by mapping them to the kernel page table, e.g., direct mapping. Blindfold takes this privilege away, by trapping any kernel attempt to configure an MMU or change a page table into Guardian.

Interrupt Table-mediated Control Flow Changes. Modern systems employ an interrupt table to mediate the change in control flow. The CPU automatically loads and executes the next instruction in memory until the current instruction triggers an exception, e.g., `svc` in ARM, or it receives an interrupt, e.g., I/O event. Upon an exception or interrupt (hereafter, we simply use the term interrupt), the CPU enters the kernel mode and jumps to the corresponding entry in the interrupt table to run its handler before returning to the original flow of execution. Because the kernel controls the interrupt table, the OS can not only access the execution context of the interrupted process but also compromise its control flow. Guardian takes control of changing interrupt tables to protect the control flow (and execution context) in interrupts. We note that different architectures use different names for the interrupt table, e.g., interrupt descriptor table (IDT) in x86 and exception table in ARM.

Architectural Support. Page and interrupt tables reside in memory. When the kernel tries to access a physical frame hosting them, the MMU consults the corresponding page table entry and checks its protection bits. By setting page table entries (PTEs) for the frames hosting page tables read-only to the kernel, Blindfold takes away the privilege of changing the page tables from the kernel. This is a technique widely used in the literature [18], [19], [20].

The above technique must be applied with techniques that deprive kernel's privilege of arbitrarily updating virtual memory control, e.g., MMU enabled and page table base registers (PTBRs), which are `CR0`/`CR3` on x86 and `SCTLR`/`TTBR`s on ARM. Modern architectures can trap updates of virtual memory control into a higher privilege level, e.g., by configuring `VMCS` on x86 and `TVM` in `HCR` on ARM, which are usually used by the hypervisor to monitor virtual machine-related events in the guest OS. Blindfold requires this widely available architectural support (§III-B).

OS Memory Optimizations. Given the central importance of memory, modern OSes such as Linux feature various optimizations. *Page migration* [11] improves memory performance of nonuniform memory access (NUMA) processors, by relocating the data closer to the processor where it will be accessed. *Demand paging* and *swapping* [21] overcome the size limit of physical memory, especially in mobile and embedded systems, by moving data between physical memory and secondary storage on demand. *Huge pages* [12] are important for supporting big-data applications by improving the efficiency of address translation. Unfortunately, previous solutions [1], [4], [22], [7] preclude such optimization opportunities by limiting the kernel's ability to manage the memory used by sensitive applications. It may lead to undesired consequence in real world scenarios. For example, without page swapping, the kernel can not swap out rarely used sensitive pages to make space when the number of free pages is low. As a result, it has to reject the following memory allocation requests until some sensitive applications finish their computing and are willing to return the memory.

Non-semantic vs. Semantic Kernel Access. Modern OSes like Linux actively take advantage of their unfettered access to user memory. The vast majority of cases are concerned with the kernel moving user-space data, e.g., demand paging and page migration. Since the kernel does not need to understand the content of the data in these cases, we call such an access *non-semantic*. We note that (*i*) the kernel always performs non-semantic accesses with the direct mapping in its kernel address space, instead of using the user page table, with the only exception of moving I/O data in I/O-related system calls, e.g., `read/write`. (*ii*) The two most popular low-level memory operations involved in non-semantic accesses are clearing a page to zero and copying a page within memory, i.e., `clear_page` and `copy_page` kernel functions.

In a minority of cases, such as system call arguments, the kernel does need to understand the data it accesses. We call such access *semantic*. We note that the kernel always performs a semantic access by dereferencing a pointer in the user address space, tagged with `__user` in the kernel source code. Due to security concerns, such dereferences always occur in a set of narrow interfaces, i.e., `copy_to_user`, and `copy_from_user` [23], [24]. This property is one of the foundations of our solution for semantic accesses (see §III-A).

Appendix A provides a more detailed study on *non-semantic* and *semantic* access in Linux.

Blindfold deals with non-semantic and semantic accesses with different mechanisms. It is worth noting that moving I/O data pointed to by the `buf` argument in `read/write` system calls is non-semantic by definition. However, Blindfold relies on end-to-end protection for I/O data (see §III-B) and thus allows the kernel to access it in the same way as semantic access.

III. Design Overview

In this section, we first introduce key insights behind Blindfold design and describe its threat model and assumptions. Then we present the overview of Blindfold in III-C.

A. Key Insights

Switching Instead of Nesting (§IV-A, §IV-C). To prevent the kernel from accessing sensitive memory with the user page table, Blindfold switches page tables instead of nesting them (Figure 1). Since nesting requires the TCB to maintain the nested (or shadow) page tables, it suffers from two problems. First, it increases the size, complexity, and attack surface of the TCB. Second, by decoupling address translation (by the OS) and protection (by the TCB), the nesting invalidates the contiguity optimizations at the OS level, e.g., a huge page in the OS while small pages in higher-privilege level. Likewise, to provide CFI in interrupts without deploying complicated interrupt tables inside the TCB, Blindfold switches the interrupt tables, using one for all unprotected processes and another (called *secure interrupt table*) for protected ones. Blindfold leverages a switching-based isolation method supported by modern architectures as introduced in §II.

Encrypted view and secure ABI for Non-semantic Access (§IV-A). Inspired by Overshadow [1], Blindfold provides the OS an encrypted view into the user space for non-semantic access, allowing it to keep managing confidential memory with all its optimizations, such as page swapping. In contrast, recent systems [4], [22], [7] chose to hide the memory regions allocated to sensitive processes or containers from the OS, which can lead to unwanted consequences in real-world scenarios, as mentioned in §II. However, encryption/decryption is expensive despite architectural extensions for cryptography on modern processors. As an optimization, we identify the most popular operations involved in non-semantic kernel access, i.e., clearing a page to zero and copying a page within memory, and let Guardian provide a secure ABI for such memory operations (see §IV-A).

Capability System for Semantic Access (§IV-B). Blindfold employs a novel and lightweight capability system to support semantic accesses of the kernel to the user space. We observe that all semantic accesses share the following three properties. First, they are *well-defined* in spatial (where) and temporal (when) boundaries. Second, the user process knows when and where the kernel accesses its address space. Third, for security reasons [24], the OS accesses the user space via a

TABLE I: Popular modern architectures support the architectural requirements of Blindfold specified in §III-B.

Architectural requirements	**x86**	**ARM**	**RISC-V**
Higher privilege mode than OS	VMX root mode	Hypervisor and monitor modes	Machine mode
Trapping virtual memory control	VMCS	TVM in `hcr` register	TVM in `mstatus` register
Invoking higher privilege mode	Hypervisor call (`vmcall`)	Trap of cache type register (`ctr`) access	Environment call (`ecall`)

set of narrow interfaces, as in both the Linux and FreeBSD kernels [23], [25]. Such interfaces are stable and can date back to the Linux kernel v2.2 and FreeBSD v2.2.1. These properties are the foundations of our solution for semantic accesses (see §IV-B). Unlike prior systems which handle semantic access, especially system calls in a case-by-case manner [1], [7], Blindfold's capability system handles all with the same design, substantially reducing the TCB size.

Architecture-agnostic Design. While recent related work has often exploited architecture-specific support, e.g., ARM TrustZone [4], [22] and Intel SGX [26], we design Blindfold to rely on only time-tested and universally available architecture features. In doing so, we aim not only to widen the user base, but also to sidestep the availability and security risks often associated with new hardware features. We report an implementation on ARMv8-A in §V, and discuss the portability to x86-based systems in §VIII.

B. Threat Model and Design Space

Threat Model. A sensitive application trusts the hardware and the Guardian. It also trusts the tools and libraries used by its developers. We assume secure boot and thus the OS and the Guardian are supposed to be initialized securely. However, we do not trust the OS or any other software at runtime, which means that the OS may be compromised after booting.

We protect the confidentiality and integrity of application data against any adversaries that can compromise the OS or access memory via DMA. We also protect the code integrity of the application and control flow integrity (CFI) across user-kernel interface, i.e., exceptions and interrupts, to fend off attacks such as return-oriented programming.

We do not protect the data sent to or received from out of a process, such as I/O and inter-process communication. Like HypSec [5] and BlackBox [7], we believe such data is better protected end-to-end [27]. We defend against replay attack only for the Guardian-encrypted data such as swapped pages by maintaining per-page signatures (§IV-A). We also defend against memory mapping-related Iago attack, which means we check the return value of system calls like `mmap` and `brk`. However, we do not defend against denial-of-service (DoS) attack. Physical and side-channel attacks are not our targets either.

Constraints. Blindfold has two absolute constraints. (*i*) It must support legacy binaries. We believe protecting application data should be transparent and orthogonal to application development and should support binaries that already exist. (*ii*) Its design must be architecture-agnostic and therefore eschew features that are available only in some specific architectures, e.g., ARM TrustZone.

Tradeoffs. We make two important tradeoffs in designing Blindfold. (*i*) We balance between changes to the OS and the size of the runtime TCB (Guardian). While it is desirable to keep both small, when we have to choose one over the other, we choose a small Guardian. Blindfold is able to achieve 2× smaller runtime TCB while requiring a similar amount of OS modifications compared to the state of the art. (*ii*) Since non-sensitive applications in the same systems may not require protection, they ideally should pay little or no performance penalty. While it is desirable to keep both small, when we have to choose one over the other, we choose a small overhead for non-sensitive applications.

Architectural Requirements. Blindfold has three requirements for the architecture. x86, ARM, and RISC-V all meet these requirements, as summarized in Table I.

A_1. A programmable higher privilege level than what the OS is running in.

A_2. A way to trap updates of the virtual memory control into the higher privilege level.

A_3. A mechanism with which an application can invoke the higher privileged software, bypassing the OS.

Additionally, we note that software-based approaches such as Nested Kernel [18] and SKEE [19] offer alternatives to implement a programmable higher privilege level without reliance on specific architectural support. These approaches can facilitate more efficient prototypes by allowing privilege level transitions to resemble kernel function calls, avoiding the overhead associated with hardware privilege level switching.

C. Blindfold Overview

Blindfold protects the confidentiality and integrity of the code and data of the application and control flow integrity (CFI) across the user-kernel interface. Its core is a small trusted software called Guardian that runs at a higher privilege level than the OS (Figure 3). The Guardian implements the four key insights described in §III-A.

Guardian protects sensitive user memory from kernel's unfettered accesses by enforcing the following invariants.

I_1. Virtual memory is always enabled after secure boot;

I_2. All page table updates must go through the Guardian;

I_3. Any kernel (or colluded user) thread can never access sensitive user space via a sensitive user page table;

I_4. Any sensitive page is either encrypted, or mapped in its associated user page table exclusively (or not mapped at all as a transient state).

With the above four invariants, Guardian guarantees that any kernel thread (or colluded user thread) can never access sensitive user pages in clear text. For semantic accesses, the OS must invoke Guardian, which verifies the capabilities before copying the clear-text data to the OS (§IV-B).

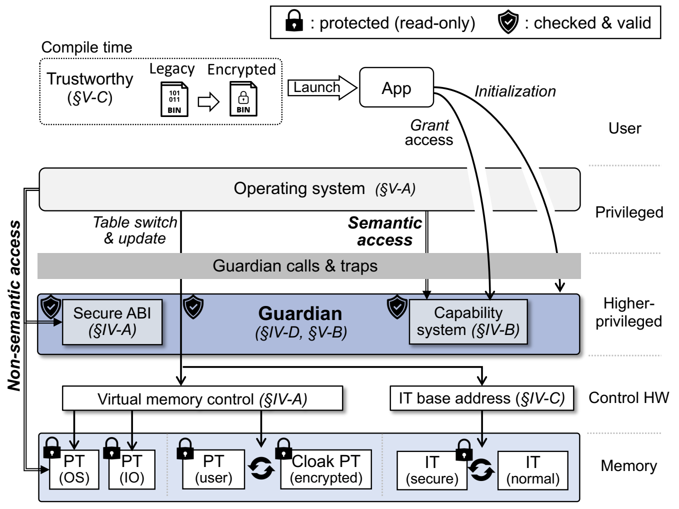

Fig. 3: Blindfold Overview. Blindfold switches the page tables (PTs) and the interrupt tables (ITs) to ensure all OS memory access (§IV-A and §IV-B) and control flow changes (§IV-C) are mediated by the TCB (Guardian) (§IV-D). For semantic access, we employ a lightweight capability system in the Guardian to verify access from the OS (§IV-B). Blindfold provides an interface to enable direct communication between the application and the Guardian bypassing the OS (§IV-D).

When a sensitive process is running, tcb forces the system to use the secure interrupt table so that it mediates interrupt handling (§IV-C) to ensure control flow integrity (CFI) in interrupts and protect sensitive process context. While the Guardian may resemble a hypervisor, it does not manage any resources or handle interrupts. As a result, Guardian remains small.

In addition to Guardian, Blindfold employs three more components to support legacy applications (see details in §V). (*i*) *Secure boot* during which Guardian enables the MMUs and invalidates the writable permission of the OS to all page tables; (*ii*) *Binary adaptation* prepares a legacy app binary for protection, encrypting loadable segments and adding helper segments; (*iii*) A small set of OS modifications so that page table updates and legitimate semantic kernel accesses must make function calls into Guardian.

IV. Design Details

We next provide details for the novel design ideas of Blindfold. (*i*) Switching page tables and providing the OS an encrypted view to regulate non-semantic kernel access (§IV-A); (*ii*) A lightweight capability system to support OS semantic access (§IV-B); (*iii*) Switching interrupt tables to protect CFI in interrupts (§IV-C). Finally, we describe the Guardian interface and security analysis.

A. Regulated Non-semantic Access

Blindfold enables performant and secure non-semantic accesses by providing secure Guardian ABI. We first explain how Blindfold disallows the OS from accessing user memory via its own page tables, instead creating an encrypted view of the user memory. We then describe the secure ABI design to optimize common cases.

Switching between user and cloak page tables. In Linux, the kernel can access the user space of a process with its user page table. Blindfold forces the kernel to use a *cloak page table* instead, when control is transferred from the sensitive process to the OS (see §IV-C). The cloak page table does not map any sensitive pages in the user space so the kernel cannot perform a non-semantic access via user space pointer dereference. We note that the kernel can still function properly with the cloak page table because it always performs a non-semantic access with the direct mapping in kernel address space (see §II). In other words, employing the cloak page table will only block illegitimate non-semantic accesses via user space pointer dereference.

Switching between clear and encrypted views mapped in user and kernel spaces respectively. At any moment, each sensitive page is either (*i*) in clear text and exclusively mapped in the associated user space, or (*ii*) encrypted, unmapped from the associated user space, and may be mapped in kernel space for non-semantic accesses (or (*iii*) in clear text but not mapped in any space as a transient state for optimization introduced later in this section).

As Guardian is invoked in every update of a page table, it can decrypt (or encrypt) a sensitive user page when the page is mapped to (or unmapped from) the user space. Meanwhile, Guardian invalidates (or validates) the PTE of direct mapping in the kernel space associated with the physical frame that hosts the sensitive page. Guardian counts mappings to each physical frame in all page tables to ensure the exclusive mapping of a sensitive page in clear text. The counting is enabled by the fact that all the updates to PTBR and a page table are trapped in Guardian.

When a sensitive page is unmapped from user space and encrypted, the Guardian generates and stores the signature of the page in a reserved region in user space (see §V-C). The signature is for verification when the page is mapped to user space again. The kernel may swap out the pages for storing signatures, in which case, the signatures of these pages are generated, and thus form a Merkle tree lazily. If the root of the tree is swapped out, its signature is stored in the Guardian.

We note that this design does not lead to race condition or frequent switching between two views in a non-semantic access. When the kernel performs non-semantic access, mostly for memory management like swapping and page migration, it will first lock the page and unmap it from the user space, to prevent modification during the memory operation like page movement. As a result, the lock and the unmapping prevent any race condition between the user process and the kernel. Such operations do not happen frequently either, assuming that the kernel performs well in its job of memory management. However, a malicious kernel can indeed slow down the process execution by maliciously performing non-semantic accesses frequently. This denial-of-service (DoS) attack does not harm the sensitive data, which we do not defend against (§III-B).

Secure ABI for memory operations. Encryption and decryption are expensive. Instead of always providing encrypted view of a sensitive page for non-semantic accesses, we identify two most popular involved operations, i.e., `clear_page` and `copy_page`, and avoid unnecessary encryption/decryption when possible.

First, in cases that (*i*) a sensitive process actively frees a virtual memory area (VMA) and (*ii*) the kernel kills a sensitive process when it finishes (or for any other reason), the kernel first unmaps the associated pages from user space and then removes the pages to zero. Without being aware that the pages are no longer used, Guardian encrypts and signs the page unnecessarily. We optimize these cases through an ABI that allows the kernel to notify Guardian that one specific VMA or all VMAs are being freed. We note that the kernel cannot abuse this ABI because the Guardian marks the VMAs are freed, unmaps and clears the associated pages in the ABI call. Any abuse will be detected when the sensitive process tries to access the pages after the call.

Second, in case of copying/moving a sensitive page from one physical frame to another, e.g., page migration, its content must be protected by encryption if the kernel performs the copying. Afterwards, the page content must be decrypted when the new physical frame is mapped to the same page. In this case, we avoid encryption/description through an ABI that triggers Guardian to copy. Guardian generates and stores the signature of the page, checks if the target frame is already mapped in any user space, and invalidates associated kernel direct mapping before performing the copy. This implies that the copied page is in clear text but is not mapped to any space. The kernel can later remap the page to a sensitive process if and only if the target process has a matched signature. We note that the kernel cannot abuse this ABI because only a verified process with the matched signature has the mapping and thus can access the copied page. The kernel can also not steal information from or fake the signature because the signature is not available for the kernel until it is encrypted and protected by the Merkle Tree in a swap-out.

DMA access. To prevent illegitimate DMA access to private data of a sensitive process, Blindfold also mediates updates to the IO page table. The Guardian only allows mappings from an IO page table to public pages, i.e., pages with `MAP_SHARED` being set in a `mmap` system call.

B. Capability-based Semantic Access

Unlike previous works in which semantic accesses are supported in a case-by-case manner, Blindfold supports all semantic accesses with the same capability system, without extra copying. We design the capability system based on the three properties shared by all semantic accesses as mentioned in §III-A. A capability is a conceptual representation of the permission to access a contiguous region in the user address space. Unlike traditional capability-based security, Blindfold's capability is stored inside Guardian, and no key is provided to the OS. The OS can request read or write access with a virtual address, and Guardian checks if the corresponding capability exists before granting access to the OS.

Each capability is specified by a 4-tuple `(addr, size, rw, life)` where the elements represent the start address of the region, the size in bytes, the permission if it is read-only or read/writable for the OS, and the lifetime of the capability, respectively. Guardian maintains a list of capabilities for each sensitive process. In our prototype, we implement it as a sorted list based on the start address and wrap it with a readers-writer lock, so it allows for concurrent read and the search time is $O(log(n))$ where n is the number of capabilities.

Capability Creation and Destruction. Blindfold does not require more information than the system call semantics and the arguments to create/destruct capabilities. To be more precise, we predetermined the required capabilities based on the semantics of system calls, e.g., how to determine the range of region and read/write permissions from the system call arguments. At runtime, the Guardian creates/destructs capabilities when a system call is trapped, based on the provided system call identifier and arguments, before/after the OS serves the system call. Specifically, for nested data structures, i.e., pointers to pointers, Guardian reads the address and length from the user space, as indicated by the pointers in the arguments, and generates the corresponding capabilities. Our current prototype supports up to three levels of pointers (§VI-D).

Blindfold categorizes capability into short- and long-lived. For almost all capabilities of which the lifetime ends with the system call returns (i.e., short-lived), the Guardian records the thread identifier (i.e., the stack pointer; see §IV-D) in `life`. Upon return of the system call, which also traps in Guardian, Guardian destructs the capability, i.e., removing it from the list. We note that `clone`, `set_tid_address`, and `set_robust_list` are the only three system calls that can create a long-lived capability that is alive until the end of the calling thread. These system calls register a 4- or 8-byte region that the kernel will access exactly once when the thread terminates. For long-lived capability, Guardian records the base address of the stack in `life`. These capabilities are disposable, which means they are destructed once they are accessed.

Capability Check and Semantic Access. The Unix-like OSes always access user space data with care, through narrow and stable interfaces such as `copy_from_user` [23]. Blindfold modifies the implementation of these interfaces so that Guardian is invoked (via `g_mov_mem` in Table II) to check capability, i.e., if any capability matches the accessing address and the read/write permission. Once Guardian verifies the legitimacy, it copies the data between the user and kernel space according to the request. Since there is no extra copying as in the buffer-based approach used in prior work [1], [4], [7] (i.e., user → buffer → kernel, instead of user → kernel), Blindfold's approach is both more efficient and more general (see comparison in §VII).

TABLE II: Guardian's ABI consists of 10 calls, which are invoked by various components in the system, including hardware, the secure interrupt table, the OS, and the `trampolines` in sensitive processes. These calls play crucial roles in regulating OS operations, maintaining OS functionality and supporting sensitive processes.

Guardian ABI	Where invoked	Role
`g_vmc_trap`	Update of virtual memory control	Trap update to virtual memory control
`g_interrupt`	Secure interrupt table	Trap when an interrupt happens to a running sensitive process
`g_set_pt`	OS	Trap when the OS tries to update a page table
`g_free_vma`		Trap when the OS reclaims the memory of VMAs
`g_copy_page`		Trap when the OS copies a page within memory
`g_move_umem`		Trap for semantic access to user space by the OS
`g_fork`		Trap when the OS forks a sensitive process
`g_proc_create`	Sensitive process (`trampolines`)	Trap when a sensitive process is being created
`g_proc_resume`		Trap when a sensitive process resumes after exception handling
`g_proc_signal`		Trap when a sensitive process is ready to handle a signal

C. Switching Interrupt Tables for CFI in interrupts

To protect the control flow integrity and execution context of a sensitive process in interrupts and exceptions, Blindfold makes the memory storing the interrupt tables read-only and forces the system to use a modified *secure interrupt table* when a sensitive process is running, which invokes the Guardian before the OS gains control. When invoked, the Guardian saves then clears the context, sets the page table base address register to the cloak page table, sets the interrupt table base address register back to the original interrupt table, and finally forwards the control to the interrupt handler.

Blindfold's design of switching between two interrupt tables contrasts those taken by prior work. For example, many [1], [5], [7] use one interrupt table in the OS to trap all interrupts into the TCB regardless of whether the running process is sensitive or not. Blindfold's design avoids unnecessary overhead for non-sensitive processes. Ginseng [20] dynamically modifies the interrupt table at *runtime* to invoke the TCB, incurring higher runtime overhead.

Resumption with Trapped Return. When the OS resumes the execution of a sensitive process after it handles an interrupt, it simply returns control to the process starting with the saved program counter; as a result, there is no obvious point that the Guardian could intervene. Blindfold solves this problem with a simple mechanism called *trapped return*. When the Guardian clears the context, it sets the return address of the interrupted process to the user-space trampoline that invokes the Guardian (`g_proc_resume` in Table II). As a result, when the OS returns control back to the process, it unknowingly invokes the Guardian. As such, trapped return does not require OS modification.

D. Guardian Design

Guardian is the software TCB running in a higher privilege mode. It is similar to a micro hypervisor, but the Guardian does not manage any resources (including nested/shadow page tables) or handle exceptions; nor does it rely on nested paging hardware support, resulting in a smaller size. Guardian protects itself by forbidding any mapping to the frames that host its own memory. This is possible because in Blindfold, the page tables are read-only to the OS and all the updates to them must go through Guardian. Guardian has a narrow ABI as summarized in Table II. Guardian is reentrant and can be concurrently invoked from multiple threads running on different cores.

Bookkeeping. Guardian tracks the necessary information of the physical frames and sensitive processes, without relying on the data structures in the OS. For each frame, it tracks whether it hosts sensitive data and a reference count across all page tables. For each sensitive process, the Guardian keeps (*i*) base addresses of the user and cloak page tables; (*ii*) addresses of the user space trampolines and signature segments; (*iii*) secret keys for cryptography; (*iv*) capability list for semantic access; (*v*) execution context when preempted; (*vi*) a list of 3-tuple `(start, end, status)` which represents the range and properties of virtual memory areas; (*vii*) signature of the root signature page. Guardian identifies a process by its page table base address and a thread within a sensitive process by the value of the stack pointer (SP). In particular, it does not trust the process/thread identifier (PID/TID) assigned by the OS.

Memory Use. The memory used for bookkeeping physical frames is about 16 MB in our prototype with 8 GB main memory. That used for bookkeeping sensitive processes is proportional to the number of sensitive processes/threads. This memory use is dominated by (*iv*) and (*v*). Since each thread has at most two long-live capabilities, while system calls are not nested and have up to six parameters, a thread requires less than 1 KB in our prototype on ARMv8-A.

E. Security Analysis and Attack Scenarios

We first analyze how Blindfold achieves the security invariants listed in §III-C (I_1 to I_4) and then discuss how Blindfold protects against popular attacks.

I_1 *Virtual memory is always enabled after secure boot.* Blindfold assumes secure boot, during which the Guardian ensures the virtual memory is enabled and properly configures the hardware to trap updates of virtual memory control (§II and §III-B). And therefore, after secure boot, the Guardian can detect any attempt of disabling virtual memory.

I_2 *All page table updates must go through the Guardian.* According to I_1, the kernel cannot bypass the virtual memory protection. So the kernel cannot modify page tables bypassing the read-only protection as Blindfold marks all page tables as read-only. The kernel cannot switch page tables without going through the Guardian either, as Blindfold traps and verifies any such attempts (i.e., updating the page table base registers; §II).

I_3 *Any kernel (or colluded user) thread can never access sensitive user space via a sensitive user page table.* The Guardian allows the use of a sensitive user page table if and only if the sensitive process, i.e., the owner of the data, has control. In all other cases, every time the control changes to the kernel or another process, the Guardian switches the page table to the corresponding one other than the sensitive user page table. Whether the change of control is triggered (*i*) by the sensitive process explicitly, e.g., a system call, or (*ii*) by an interrupt from outside the process, the Guardian always gains control before the switching via the secure interrupt table and ensures that the appropriate page table is used (§IV-C).

I_4 *Any sensitive page is either encrypted, or mapped in its associated sensitive user page table exclusively (or not mapped at all as a transient state).* All sensitive pages are originally encrypted. The Guardian ensures that only an exclusive mapping exists for a sensitive page during and after transitions from an encrypted view to a clear view (and vice versa). Specifically, the Guardian decrypts the pages only after invalidating the corresponding kernel direct mappings and ensuring that the associated frames are mapped exclusively to the sensitive user space (§IV-A). We note that, by leveraging I_2, the Guardian maintains a complete list of any existing mappings in the page tables. The Guardian does the opposite for the transition from a clear view to an encrypted view. The transient state does no harm to confidentiality since it does not have any mappings or allow any access (§IV-A).

We note that I_3 and I_4 guarantee the kernel (and colluded user processes) can never access a sensitive page in clear text.

Attack Scenarios. We discuss how Blindfold protects against popular attacks. We note Blindfold does not defend against Denial-of-service (DoS) attacks as pointed out in §III-B, e.g., slowing down the process execution by maliciously performing non-semantic access frequently as discussed in §IV-A.

Replay attacks: The OS can not replay encrypted data since the Guardian maintains and checks the per-page signatures (§IV-A). Even if the OS swaps out a page containing the signatures, the Guardian encrypts it and generates signature of the encrypted page, effectively forming a Merkel Tree.

Return-oriented programming (ROP) attacks [28]: The OS can modify the return address to bypass trapped return (§IV-C) and perform a ROP attack. However, the compromised process immediately “loses” its sensitivity — it can no longer access the sensitive context or switch to its user page table. As a result, it will not be able to access its own memory. That is, even a successful ROP attack will never compromise any sensitive data.

Iago attacks [29]: The OS may misbehave in interrupt and system call handling. For example, it may return the address of the stack in a `mmap` system call, tricking the process to overwrite its stack unknowingly. Blindfold fends off such memory-mapping Iago attacks by maintaining its own bookkeeping of memory frames and per-process virtual memory areas. Like BlackBox [7], Blindfold defends against a stronger threat model than TrustShadow [4], which protects both sensitive pages and page tables by hardware.

V. Implementation

To validate Blindfold, we prototype it with Linux kernel v5.15 on ARMv8-A architecture. Specifically, we implement the Guardian on top of ARM Trusted Firmware [30] for ARMv8-A. We note that Blindfold has a smaller, i.e., about a half of, runtime TCB and requires a similar amount of kernel modifications compared to recent related systems [4], [7] that also support legacy binaries. Beyond the kernel modifications (§V-A) and the Guardian (§V-B), Blindfold requires binary adaptation to support legacy binaries (§V-C).

A. Linux Kernel Modification

Blindfold introduces about 400 LOC to the Linux kernel, which is close to that in TrustShadow [4] (0.3K) and Black-Box [7] (0.5K). Next, we describe some key changes.

Trapping Page Table Updates. Blindfold requires that all page table updates be trapped into the Guardian. To achieve this, Blindfold modifies the implementation of low-level stable kernel interfaces like `set_pte` to add trampolines to the Guardian via ABI `g_set_pt`. We note these interfaces have been stable since Linux kernel v2.0 and the modification is about 50 LOC.

Cloak Page Table for Blocking User Space Pointer Dereference. Blindfold disallows the kernel to directly dereference a user space pointer in order to access sensitive user pages. To achieve this, Blindfold modifies the kernel in three places to leverage the cloak page table. First, we add a new field, `c_pgd`, to the memory descriptor `mm_struct` to store the base address of the table of pages cloak. Second, we modify the kernel to create (destroy) a cloak page table for a sensitive process when the process is created (terminated). When Guardian is invoked by `g_proc_create` and `g_fork`, it ensures that the cloak page table is read-only for the kernel, before allowing the sensitive process to start (§V-B). Finally, we modify the page fault handling logic so that it invokes the Guardian through `g_set_pt` to update the cloak page table if the page fault happens in the `trampoline` segment (see §V-C), which is the only page that is mapped in the cloak page table and invokes the Guardian in *trapped return* (§IV-C). These involve modification of about 40 LOC.

Secure ABI for Optimizing Confidential Memory Management. To reduce the overhead of unnecessary encryption/decryption for confidential memory management, we modify kernel functions such as `unmap_vmas`, `migrate_page_copy` and `do_cow_fault` to invoke Guardian via `g_free_vma` and `g_copy_page`. So, the kernel does not actually perform the non-semantic access via low level interface like `clear_page` and `copy_page`. Instead, Guardian does it on behalf of the kernel. This incurs about 60 LOC of kernel modification.

Semantic Access. To semantically access user data, we modify the implementation of intra-kernel interfaces like `copy_from_user` [23], which have been stable since Linux kernel v2.2. The modification invokes Guardian with `g_mov_umem` to check capability and then perform the copy on behalf of the kernel (§IV-B). This incurs about 40 LOC of kernel modification.

Beyond the above modifications, we also implemented an optimization to group small protected pages into a few huge pages, to minimize the impact of breaking down the kernel direct mapping in huge pages. The rest of the modification is mainly comprised of a general interface to make Guardian ABI call written in inline assembly and some macro defined in header files.

B. Guardian Implementation

We implement Guardian on top of the ARM Trusted Firmware [30] that runs in EL3 on ARMv8-A. Our prototype consists of about 2.2K LOC in Rust for high-level logic and about 0.2K lines of assembly for switching between worlds. The Guardian additionally employs the cryptography libraries (13K LOC) from the RustCrypto project [31], `linked_list_allocator` crate (1.2K LOC) and the readers-writer lock from the `synctools` crate (0.7K LOC). We next describe a few key implementations.

Page Table Control. Blindfold traps updates to (*i*) the virtual memory control and (*ii*) the page tables into the Guardian.

For (*i*), the Guardian sets the *TVM* bit in `HCR_EL2` on ARMv8-A (Table I). As a result, when the kernel updates virtual memory control registers such as `SCTLR_EL1` and `TTBRx_EL1`, the hardware will trap into EL2 where Guardian ABI `g_vmc_trap` (Table II) will be invoked. Guardian updates the virtual memory control on behalf of the kernel after checking that the update does not violate the invariants in §III-C. Specifically, the kernel is not allowed to switch the kernel page table by updating `TTBR1_EL1`. When the kernel attempts to update `TTBR0_EL1`, if the page table is a sensitive user page table, Guardian will replace it with the corresponding cloak page table. The Guardian also mediates updates to address space identifier (ASID) to prevent the kernel from accessing sensitive user space through cached TLB entries tagged with the user-space ASID, bypassing the need for a TLB flush. On the other hand, if it is the first time the Guardian processes the page table, i.e., a new process, the Guardian will walk the entire page table to mark page table pages as read-only, count page references, and invalidate any mappings to protected pages, e.g., sensitive pages in clear-text, secure interrupt table, and page table pages, before it returns control back to the kernel. These are enabled by the Guardian's bookkeeping about the physical frame status (§IV-D).

For (*ii*), the kernel invokes Guardian through `g_set_pt` as described in §V-A. In a trap, Guardian walks the subtree associated with the target page table entry. In the page table walk, Guardian marks page table pages as read-only and refuses any mappings to decrypted sensitive pages or writable mappings to secure interrupt table and page table pages. As a result, Guardian guarantees that all page table pages are read-only to the kernel even if the page tables are growing.

Interrupt Table Control. Blindfold protects the secure interrupt table in the same way as it protects the page tables. The secure interrupt table is a wrapper of the original interrupt table, where each entry includes an instruction to invoke Guardian's ABI `g_interrupt` before jumping to the original interrupt handling logic. To be more precise, in ARMv8-A, we add a `smc` instruction as the first instruction for all entries, followed by the original interrupt handling logic. As a result, by switching to using the secure interrupt table while a sensitive process is running (§IV-C), we can invoke the Guardian to mediate control flow changes due to system calls and interrupts for the sensitive processes. We note that the kernel cannot update the interrupt table base address register while a sensitive process is running. Moreover, table switching does not require a TLB flush because Blindfold uses different kernel virtual addresses for the two tables.

Page Fault in Semantic Access. When being invoked by `g_move_umem` and handling semantic accesses, Guardian may trigger page faults when it copies the user-space data on behalf of the kernel. In this case, Guardian delegates the page fault to the kernel and applies the trapped return (§IV-C). More precisely, Guardian sets the return address, i.e., the `ELR_EL1` register on ARMv8-A, to where the kernel makes the `g_move_umem` ABI call, so that the kernel will retry the semantic access after handling the page fault.

Invoking Guardian. Table II summarizes the scenarios and methods for invoking Guardian. For calls originating from the OS or the interrupt table, a single `smc` instruction suffices to change the privilege level. However, for invocations from sensitive processes, a direct transition from EL0 to EL3 is not feasible, as no instruction supports this transition. To address this problem, the corresponding trampoline in a sensitive process triggers a trap into EL2 by accessing `CTR_EL0` and then immediately invoke Guardian with a `smc` instruction. We note that `HCR_EL2` needs to be properly configured for this trapping. This method leverages EL2 as a bridge to invoke Guardian. This is quite similar to how the kernel's update to the virtual memory control is trapped into Guardian, described above.

C. Sensitive Process Execution

Next, we describe how Blindfold works during the life cycle of a sensitive process.

Binary Adaptation. Blindfold supports legacy binaries, but must adapt them for secure execution. Blindfold assumes the developer and Guardian have their own public-private key pairs. At compile time, the developer prepares the binary via the Binary Adapter. The Binary Adapter generates symmetric keys to encrypt and sign loadable segments and metadata such as binary headers. It encrypts the symmetric keys with Guardian's public key and embeds them into the binary along with their signatures so that Guardian can check the integrity of the keys and the segments. We store all signatures in an added `signature` segment and apply the same adaptation to library binaries if the app is dynamically linked. We also reserve a virtual memory area in the form of a `BSS` segment to store heap and stack signatures at runtime.

The Binary Adapter also adds a text segment with the three user-space trampolines (Table II) to the binary and redirects the entry point (i.e., `_start`) to one of the trampolines (`g_proc_create`). This `trampoline` segment only contains a few instructions, smaller than the shim introduced in Overshadow [1] by three orders of magnitude. We discuss the use of each trampoline in detail in the following.

Sensitive Process Creation. When the kernel transfers control to the entry point of a process as the last step of process creation, it unknowingly transfers to the `trampoline` segment to invoke Guardian via ABI `g_proc_create` (see trapped return in §IV-C). The Guardian checks the integrity of the metadata with the Binary Adapter's public key and extracts the symmetric keys with its private key. With the headers, Guardian walks the list of virtual memory area descriptors in the kernel space to learn the virtual addresses of the segments. Then it walks the user page table. If a page is present, Guardian hides the mapped frame from the kernel, checks its integrity, and decrypts it with the extracted keys. Finally, Guardian configures the interrupt table base address register to the secure interrupt vector table (§IV-C) before returning control to the original entry point.

Clone and Fork. Guardian can identify `clone` and `fork` system calls from the stored context (§IV-C). The Guardian is aware that the stored context will be restored twice if the system call succeeds (for the parent and the child). When a cloned child thread starts with `g_proc_resume`, the Guardian recognizes the thread and restores the context accordingly, based on the sensitive process identifier and the stack pointer (§IV-D). As for a forked child process, we modify the kernel to allocate a new cloak page table and invoke Guardian's ABI `g_fork` by the end of the fork system call handling. So Guardian can identify the forked process and restore context accordingly when it returns from the handling.

Signal Handling. There are two challenges in signal handling: (*i*) execution environment setup and (*ii*) control transfer.

For (*i*), the Linux kernel reuses the user stack (or an alternate signal stack) to execute the signal handler, and thus writes to the user space with `copy_to_user`. We add a capability starting at the top of the stack whenever an interrupt happens and destruct it before resuming process execution, since signal handling can happen at any time when the control returns from an interrupt.

For (*ii*), Guardian can identify system calls like `sigaction` and learn the addresses of legitimate signal handlers from the parameters when the system calls are trapped in §IV-C. It redirects the pointer of the signal handler in parameter to the trampoline that invokes `g_proc_signal` (see trapped return in §IV-C). Therefore, Guardian is invoked in all attempts to run a user-defined signal handler. At which point, it transfers control to the original signal handler after verifying the legitimacy. As a result, Blindfold can prevent malicious control transfer to execute sensitive process execution.

TABLE III: List of micro and macro benchmarks

Name	Description
LMbench	`LMbench` v3.0-a9 [32] micro benchmarks
OTP	One-time password generator (`OTP`) from open source code of Ginseng [20]
DNN	Deep neural network-based object classification (`DNN`) from open source code of OpenCV [33]
Redis	`redis` v7.2.5 using the `memtier` benchmark v1.2.0 with default `redis` protocol
Memcached	`memcached` v1.6.9 using the `memtier` benchmark v1.2.0 with `memcache_text` protocol
Nginx	`nginx` v1.20.1 server handling 100 concurrent requests from remote `ApacheBench` v2.3 client
Apache	`apache` v2.4.46 server handling 100 concurrent requests from remote `ApacheBench` v2.3 client

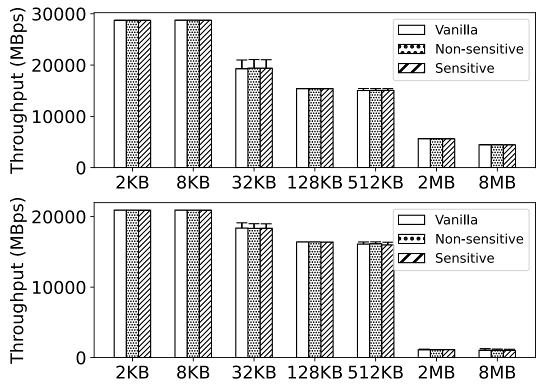

Fig. 4: LMbench memory throughput benchmark: read (top) and write (bottom). The overhead of accessing application's own memory is negligible in Blindfold.

VI. Evaluation

We quantify Blindfold's performance for both sensitive and nonsensitive configurations compared to that on vanilla Linux with both micro and macro benchmarks. We focus on the runtime overhead because the overhead of secure boot and binary adaptation is one-time and small.

We run the benchmarks listed in Table III. For all evaluations, we run Blindfold on a Raspberry Pi 4 Model B, which is equipped with Quad core Cortex-A72 at 1.8GHz and 8 GB DRAM. For those benchmarks that involve both servers and remote clients, the servers are running on a Raspberry Pi 4 Model B, while the clients are running on an Intel NUC13ANHi7. The two machines are connected directly with an Ethernet cable.

A. Micro benchmark

Negligible overhead for application's own memory access. Blindfold aims to impose as little overhead as possible for the common case in which a sensitive process accesses its own memory. In Figure 4, the memory throughput benchmark from LMbench [32] shows such overhead is negligible.

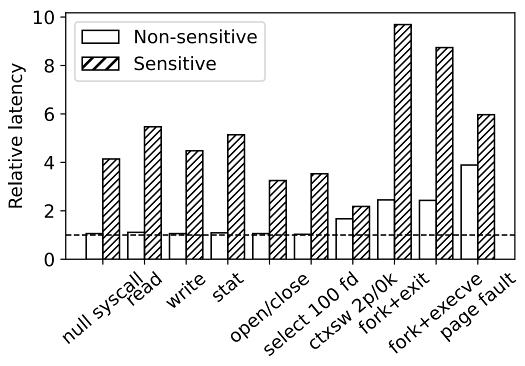

Fig. 5: LMbench latency micro benchmarks. The two bars for each benchmark represent the latency ratio on Non-sensitive and Sensitive configurations compared to the latency on Vanilla Linux, indicated by the horizontal dashed line.

We further analyze the sources of overhead introduced by Blindfold using more LMbench benchmarks. Figure 5 presents the results by comparing the latency of non-sensitive and sensitive configurations with that on vanilla Linux.

Overhead for system calls. System call related benchmarks, including *null syscall*, *read*, *write*, *stat*, *open/close* and *select*, introduce negligible overhead to nonsensitve applications, i.e., the ratios are close to 1. On the other hand, the latency of sensitive applications is 3 to 5×, which includes the overhead of trapping the system call to store sensitive context and switching tables, as well as the overhead of semantic access in system call handling.

Page table update overhead anaysis. The other four benchmarks indicate that *context switch*, *page fault*, *fork* and *execve* impose non-negligible overhead on nonsensitive applications, because all these involve trapping into Guardian for security checks. A *context switch* necessitates an update to the page table base register, which is trapped by hardware into Guardian. For each *page fault*, the kernel also must call the Guardian to update the page table. As for *fork* and *execve*, Blindfold does not trap these two events explicitly for non-sensitive processes. However, both benchmarks create a new process, which implies page table updates when creating the page table for the new process.

On the other hand, Blindfold imposes a higher overhead on sensitive applications. We classify the benchmarks into two groups and analyze them one by one. First, *context switch* introduces overhead due to the trapping of page table base register updates as mentioned above. It is slightly higher than that on nonsensitive applications because Guardian has to look up its bookkeeping for the address of the cloak page table for the update. Second, for those benchmarks involving page table updates, including *fork*, *execve* and *page fault*, they introduce overhead due to the trappings of the page table updates, as mentioned above. However, the overhead is much higher because (un)mappings for a sensitive process involve cryptography such as encryption/decryption and signature generation/verification. As the cryptography operations are expensive, the overhead is much higher than that on a nonsensitive process.

To conclude, the major source of overhead for non-sensitive applications is page table updates. For sensitive applications, the overhead of page table updates and the related cryptography operations is high. Beyond that, overhead introduced by system calls and semantic accesses is non-negligible.

B. Macro benchmark

Micro benchmarks (§VI-A) indicate that Blindfold imposes high overhead on an application when it calls *fork* and *execve*. So in the following macro benchmarks, we first present and analyze the results from short-lived applications that *fork* and *execve* from binaries. Then we present results of the long-lived daemon applications.

1) Short-lived applications We use the two open source applications, i.e., one-time password generator (`OTP` from Ginseng [20]) and DNN-based object classifier (`DNN` from OpenCV [33]), for our evaluation. These are popular applications in real world scenarios and both are sensitive as the users care about the confidentiality of the input/output.

Cryptography is the major source of overhead for short-lived application. The `Original` columns in Table IV presents the results of `OTP` and `DNN` benchmarks, measuring the average latency of 1000 executions with `time.perf_counter` from Python. The overhead of non-sensitive `OTP` and `DNN` are 65.6% and 22.8% respectively, while the latency of sensitive ones are 2.8× and 9.1×. The reason why the sensitive `OTP` has less overhead, i.e., 2.8× instead of $9 \sim 10\times$, is because `OTP` is compute-intensive and does not require much memory. So it does not trigger as many page faults and cryptography operations as `DNN`. In contrast, a cold-started `DNN` consumes more pages and necessitate page table updates that involve expensive cryptography.

Performance optimization for short-lived applications. The high overhead of cryptography imposes a formidable cost to create a sensitive process: the cold start of the program causes a lot of page faults (and cryptographic operations). This suggests that Blindfold is more efficient at protecting long-lived services than ephemeral ones. For example, our original implementations (`Original`) of `OTP` and `DNN` create a new process for each authentication and inference request, respectively. After we slightly modify them into long-lived applications (`Optimized`), the overhead from Blindfold drops, as shown in `Optimized` columns in Table IV.

2) Long-lived applications We use four applications that are popular in real-world scenarios for our long-lived application benchmarks. `Memcached` and `Redis` are in-memory key-value databases that are memory intensive, while `Nginx` and `Apache` are web servers that are I/O intensive.

Figure 6 presents the relative latency of nonsensitive and sensitive configurations, noting that the x-axis represents the latency in vanilla Linux and the y-tick starts from 1. We also note that even if we use the same version of `memtier` to

TABLE IV: Latency of **OTP** and **DNN**. The three rows represent latency of vanilla, non-sensitive, and sensitive executions. The "Original" and "Optimized" columns represents latency of executions before and after optimizing the short-lived applications into daemons.

Benchmarks	**OTP** (μs)		**DNN** (ms)	
	Original	Optimized	Original	Optimized
Vanilla	2681.58	43.27	1113.82	581.64
Non-sensitive	4442.00	46.43	1368.04	597.51
Sensitive	7399.24	62.51	10121.84	640.60

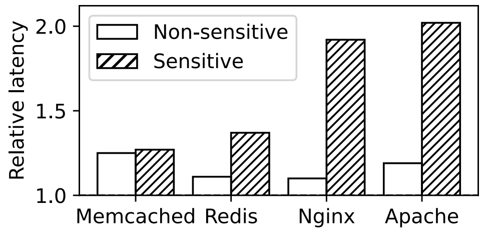

Fig. 6: Long-lived application macro benchmarks. The two bars for each benchmark represent the latency ratio on Non-sensitive and Sensitive configurations compared to the latency on Vanilla Linux, indicated by the x-axis (y-tick starts from 1).

benchmark both `Memcached` and `Redis`, the parameters are different since the two applications use different protocols. The results show the overhead varies from one application to another and we next analyze them case by case.

Long-lived applications trigger few cryptography operations. The overhead of both sensitive and non-sensitive `memcached` are high but close, which implies that the major source of overhead is page table updates that do not involve cryptography. Unlike a cold-started application, most pages of a long-lived application are already decrypted. As a result, most pages will remain decrypted as long as the kernel does not perform non-semantic access. To be more precise, while there are not many non-semantic accesses such as page swapping, (*i*) the kernel allocates anonymous pages to the application as its working set grows large, or the kernel reclaims page when the application frees the pages; (*ii*) the kernel may migrate pages via secure ABI. Neither of the cases involves encryption/decryption.

System calls are the major source of overhead for long-lived application. While both `Memcached` and `Redis` are in-memory key-value databases, they do not share the same overhead pattern. We investigated the execution of `Redis` and found that it makes frequent system calls. Surprisingly, about half of the system calls are `gettimeofday`. We note that we are not the first to discover this issue, as it is already reported in Redis's GitHub repository [34], [35]. Such system calls are not necessary for in-memory key-value databases since they do not require a very fresh time [35]. This suggests that the higher overhead of `Redis` can be optimized away.

On the other hand, as IO-intensive web servers, `Nginx` and `Apache` make more frequent system calls, such as `epoll_wait` and `recv_from`, which is the main reason why they both

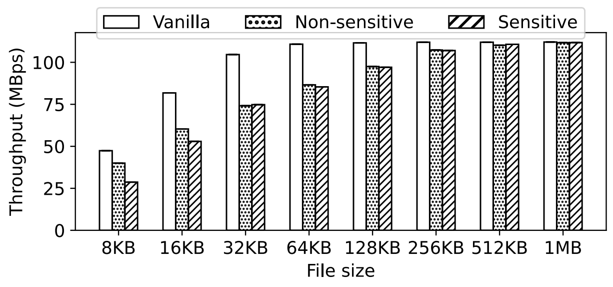

Fig. 7: Nginx throughput for serving an HTML file repeatedly over a local-area network (Ethernet). Since most of the overhead is fixed, the relative overhead of Blindfold diminishes as the file size increases.

have a much higher overhead in the sensitive configuration.

Performance optimization for long-lived applications. Since system calls are a major source of overhead for long-lived applications, a straightforward optimization is to eliminate unnecessary system calls, such as the frequent use of `gettimeofday` in Redis [34], [35].

Beyond that, another optimization is to batch multiple system calls into one, if possible, or to fully utilize each system call. For example, sending a large file instead of several smaller ones with `Nginx` is more efficient. Figure 7 shows that the overhead diminishes as the file size increases. We note that in the evaluation shown in Figure 7, we do not encrypt the file to better demonstrate the overhead imposed by Blindfold.

Performance Comparison with Related Systems. The most related recent systems are BlackBox [7] and TrustShadow [4]. Unfortunately, an apple-to-apple performance comparison with any of them turns out to be rather difficult. First, they are closed-source and/or outdated, thus a fair direct comparison is not practically possible. Second, the systems require different hardware due to their varied goals and implementations, preventing us from comparing them under the same controlled hardware and environment setup. We will discuss the differences in the next section (§VII).

C. Hardware impact on performance

Blindfold's overhead highly depends on the silicon, because it involves frequent switching among three privilege levels for user, kernel, and Guardian, respectively. We devise a nano benchmark to measure the latency of such privilege level switching on Raspberry Pi 4 Model B, the system used in the reported evaluation, and Hikey 960, a system used in an early prototype of Blindfold reported in [17]. We find that the cost of switching can be very different, as shown in Table V and Table VI. In particular, switching involving EL2 and EL3 is very expensive, which to some extent explains the high overhead introduced by system calls in our implementation (Figure 5). Moreover, because there is no instruction that can directly trigger a trap from EL0 to EL3 on ARMv8-A, our implementation uses EL2 as a stepping stone into Guardian (in EL3) from a sensitive process (§V-B). This further contributes to the overhead. All these suggest that we should implement

TABLE V: Raspberry Pi 4 Model B: Overhead in # of cycles for CPU privilege mode change from Row to Column.

Mode	EL0	EL1	EL2	EL3
EL0		262	975	N/A
EL1	59		1046	273
EL2	65	59		307
EL3	66	58	283	

TABLE VI: Hikey 960: Overhead in # of cycles for CPU privilege mode change from Row to Column.

Mode	EL0	EL1	EL2	EL3
EL0		44	868	N/A
EL1	31		1308	43
EL2	248	16		817
EL3	17	36	634	

Guardian in EL1 using software-based privileges such as those featured in [19], [18].

D. Kernel functionality

In this section, we show that the Linux kernel can fulfill its functions under the restrictions imposed by Blindfold. Not all of these functions are supported in related systems.

System Call Coverage. We evaluate system call coverage in Blindfold with the Linux Test Project (version 20230516) [36] test cases for system calls. We run the adapted benchmark binaries as sensitive applications in Blindfold and compare the results with that of running the vanilla benchmarks on vanilla Linux. There are 1340 test cases, 197 of which are not supported by vanilla Linux on Raspberry Pi 4 due to different kernel configuration or missing hardware/OS features. On vanilla Linux, 1065 test cases pass, while the remaining 78 fail. With Blindfold, 1035 test cases pass, which is more than 95% of those that also pass on vanilla Linux. In comparison, BlackBox passes about 90% of those that pass on vanilla Linux [7], while TrustShadow did not report their coverage.

We analyze the system calls that Blindfold failed to support and identify the reasons for these failures. In summary, these system calls either require trust beyond the boundary of a single process (e.g., identifiers for shared memory regions) or rely on kernel/driver-specific knowledge.

- `process_vm_readv` and `process_vm_writev`: These two system calls transfer data between processes identified with a provided `pid`, while Blindfold cannot trust the `pid` as an identifier (§IV-D).
- `shmat` and its relatives: These system calls use `shmid` to identify a piece of shared memory, which is neither trusted nor recognized by Blindfold.
- `ioctl`: Blindfold cannot fully support it as it relies on the device driver-specific semantics.
- `io_uring` and `vmsplice`: We defer supporting system calls that require either very complicated internal structures such as pointer in pointer, especially more than 3-levels, or in-kernel state that is not known to the Guardian.
- We omitted other minor test cases as they do not significantly affect Linux's functionality, e.g., `profil` system call.

Non-semantic Access. We evaluate non-semantic access using the `migrate_pages` system call. `migrate_pages` triggers the OS to migrate all pages of a process from one memory region to another. Because the `migrate_pages` handler avoids moving pages within the same memory region, we modify the kernel to bypass this optimization since the Raspberry Pi 4 does not have multiple memory regions (NUMA).

In more than 1,000 page migrations, all page movements are successful, and the sensitive process continues to run correctly. We note that recent related systems, such as TrustShadow [4] and BlackBox [7], do not support the non-semantic access necessary for page migration because they hide memory allocated to secure processes/containers from the OS.

Supporting Existing Binaries with Adaptation. Blindfold supports existing binaries only requiring adaptation (§V-C). We adapted all the applications for micro and macro benchmarks used in our evaluation.

VII. Related Work

Virtualization-based Solutions. Overshadow [1] employs a hypervisor as the TCB to control user data access from the untrusted OS using nested virtualization (Figure 1). Although often justly criticized for its massive TCB [7], Overshadow pioneered two ideas. First, goal-wise, it provides the OS an encrypted view for non-semantic access to user space. Although many later systems eschew this goal [4], [7], having an encrypted view is essential to allow resource management inside the OS for a protected process. Second, solution-wise, it pioneered the use of nested virtualization to protect applications from the underlying untrusted OS, which is adopted by many later systems [5], [6], [7]. While Blindfold shares the same concept of encrypted view, Blindfold uses switching, instead of nesting (§III-A). Most recently, *BlackBox* [7] further innovates the use of nested page tables by substantially reducing the TCB running in the hypervisor mode and expanding the protection to containers. BlackBox, however, does not allow non-semantic accesses by the OS; once a physical memory frame is allocated to a protected container, it disappears from the OS's view. As a result, many OS functions will stop working for these frames, e.g., swapping, memory compression [15], and page migration including related signals such as `move_pages` and `migrate_pages`.

Hardware-based Solutions. Instead of using the hypervisor mode for TCB, another line of work takes advantage of architecture-specific hardware features. *TrustShadow* [4] runs a protected process inside the ARM TrustZone's Secure World that is isolated from the OS in the Normal World by hardware. The memory used by the protected application is allocated in the Secure World and as a result, disappears from the OS, just like in BlackBox. Other hardware-based solutions, such as Intel SGX, depend on architecture-specific hardware implementations, making them difficult to deploy on other architectures [37], [38].

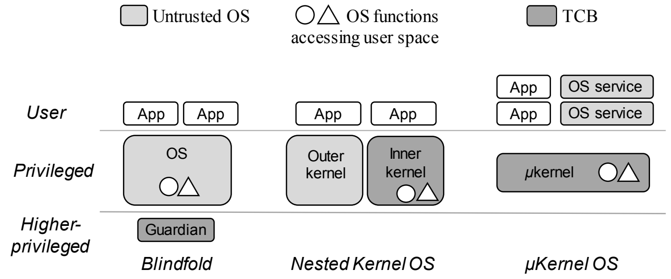

Fig. 8: Nested kernel and microkernel-based OSes limit OS access to the user space by moving OS functions outside the TCB. However, OS functions requiring user-space access must reside in the TCB, increasing the TCB size.

Without systematically addressing access to user space by the OS, these works rely on case-by-case solutions that are incomplete and inefficient. For instance, since the OS cannot access the user space, the TCB needs to prepare a buffer to store syscall arguments so that the OS access them from the buffer. As a result, the TCB not only copies data between the OS and the user application, but also manages the memory used by the buffer, which can be arbitrary large for some system calls, e.g., `read()/write()`. In addition, buffer-based data copy cannot support `futex` which requires atomic operation of data copy and wait queue update. Existing work proposed another workaround to address this issue, e.g., a `futex` syscall handler inside the TCB [1] and modification to the `futex` handler in the OS [7].

Paravirtualization-based Solutions. Blindfold bears similarity to *paravirtualization* in which a VMM works with a modified guest OS and/or augmented applications making hypercalls. However, unlike the existing VMMs that protect user-space, e.g., *InkTag* [2] and *Sego* [3], Blindfold pursue different goals resulting in different design choices: (1) reducing the TCB size, much smaller than existing VMMs, and (2) preserving optimizations in modern OSes such as page migration, instead of nesting page tables via virtualization, which effectively nullifies the OS-level optimizations.

Comparison with Related OS Designs. Exokernel [39] and *μKernel* OSes can also prevent OS services from accessing user memory by relocating them to the user space. However, without answering how OSes can manage memory and serve system calls without unfettered access to the user space, these functionalities have to stay in the TCB (Figure 8). *Nested Kernel* [18] provides a mechanism to implement privilege in software. Blindfold can use such a mechanism in place of the architecture requirement (A1 in §III-B). Indeed, even though Nested Kernel does not aim at protecting user address space, one can disable the outer kernel from accessing the user space and let the nested kernel control page-table updates to protect the user space. However, without access to the user space, the outer kernel cannot anymore manage user pages or handle system calls, resulting in a larger inner kernel or TCB as shown in Figure 8.

Discussion on TEE and Confidential VMs. Compared to TEEs that rely on hardware features such as ARM TrustZone, Blindfold offers a software-driven solution with broader applicability. In particular, certain concepts from Blindfold could also enhance existing hardware-based solutions. For example, a Realm management monitor in ARM CCA [40] can apply Blindfold 's capability system design to serve the VMs in Realm. In addition, since Blindfold is orthogonal to confidential VM solutions such as Intel TME-MK [41], Blindfold can improve security *inside* a confidential VM, between the OS and applications.

VIII. Discussion

We next discuss (1) how we implement a basic x86-64 port (without full features), (2) some unsolved problems in Blindfold design and new opportunities.

Support for x86-64. We have implemented Blindfold on x86-64 as a proof of concept and report the two key differences from that on ARMv8-A. The x86-64 Guardian is implemented as an extension of the micro hypervisor Bareflank [42]. Although our x86-64 prototype uses the Host mode to run Guardian, we do not use nested paging or EPT. Next, we present some key differences from the ARMv8-A prototype.

First, x86 requires a different hardware trapping mechanism, as mentioned in §II and summaried in Table I. To trap virtual memory control on x86-64, Guardian sets the *CR3-load exiting* bit and the *MSR bitmaps* of `EFER` in Virtual Memory Control Structure (VMCS), as well as configures the *Guest/Host Masks* and *Read Shadows* for `CR0` and `CR4` accordingly.

Second, x86 requires a different interrupt mechanism because it has a more strictly structured interrupt table. On ARMv8-A, the interrupt/exception table consists of 16 entries and each entry contains up to 32 instructions. On x86-64, an interrupt table is a jump table where each entry is a single 64 bit address indicating the target interrupt handler. Therefore, unlike ARMv8-A, we cannot directly insert a guardian call into each entry. We resolve this issue with indirection. Specifically, the x86-64's *secure interrupt table* points to a "wrapper table" of which each entry contains only two operations, a guardian call, i.e., calling the Guardian's ABI `g_interrupt`, and a jump instruction back to the original interrupt handler.

One complication arises as x86-64 supports fast system calls with the `syscall` instruction, which jumps to the system call handler pointed by the `lstar` register, bypassing the interrupt table. We solve it by modifying the `lstar` register to point to another wrapper, which jumps to the original address stored in `lstar` after invoking the Guardian.

Although not all features, such as secure ABI for optimizing non-semantic accesses, have yet been supported in our x86-64 prototype, we believe it adequately demonstrates that Blindfold 's design can be applied to x86-64.

Formal verification of the implementation. Although the Blindfold design provides security properties as shown in §IV-E, the implementation may contain deviations from the design that can lead to a new attack surface. Formal verification methods can be used to address this problem by mechanically checking for gaps between the design specification and the implementation [43], [44]. Since formal verification involves significant overheads in many cases, especially in terms of developer hours required to describe specifications and invariants, we leave this for future work.

Side Channel. In this paper, we have not considered side-channel attacks in Blindfold. There are two side channels that we should investigate. First, the OS still knows what system calls are made and with what arguments. It may be able to infer about a sensitive process based on the system calls made. Second, while the OS cannot modify a page table or read a user page in Blindfold, it can infer about which user pages are accessed and when because it still handles page faults.

Memory Compression. The OS requires semantic accesses to memory in order to compress them effectively. The encrypted view supported by Blindfold will render compression ineffective. As a result, memory compression [15] is the only kernel memory optimization that loses its effectiveness under Blindfold, although it still works. To support memory compression in Blindfold, the application can explicitly grant the OS the capability to gain semantic access to the pages to be compressed. This requires the application developer to modify their applications accordingly.

Optimizing Blindfold for I/O-intensive Workload. We shows with Nginx (Figure 7) that Blindfold introduces a fixed overhead per system call independent of the size of data being served. Therefore, I/O-intensive applications may want to amortize the overhead, for example, by using multi-message system calls such as `recvmmsg` or `sendmmsg`. However, Blindfold currently requires the OS to use one Guardian ABI call to copy each packet into the protected memory of the process. One way to overcome this limitation is to improve the Guardian ABI to support batching. With that, the OS can employ the scatter-gather pattern to collect multiple packets per Guardian ABI call.

IX. Conclusion

Modern operating systems (OSes) assume that applications trust them, granting the OS unfettered access to any data in user applications. Blindfold demonstrates that such unrestricted memory access is not fundamentally necessary for the OS to perform its tasks, including memory management. We implemented a prototype of Blindfold on ARMv8/Linux, leveraging a tiny trusted program, called Guardian, running at a higher privilege level to mediate memory access and exception handling by the OS. We evaluated Blindfold 's performance using macro and micro benchmarks, observing that Blindfold provides competitive performance with a runtime TCB that is 2 to 3× smaller compared to previous work.

Acknowledgments

This work is supported in part by NSF Awards #1730574, #2130257, and #2112562. The authors are grateful to Dr. Min Hong Yun for his contribution during the early stage of the project.

References

[1] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2008.

[2] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.

[3] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficiently verified untrusted system services," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2016.

[4] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2017.

[5] S.-W. Li, J. S. Koh, and J. Nieh, "Protecting cloud virtual machines from hypervisor and host operating system exploits," in *Proc. USENIX Security Symp.*, 2019.

[6] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "TwinVisor: Hardware-isolated confidential virtual machines for ARM," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2021.

[7] A. Van't Hof and J. Nieh, "BlackBox: A container security monitor for protecting containers on untrusted operating systems," in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2022.

[8] M. Logan and P. Kinger, "Linux Threat Report 2021 1H: Linux threats in the cloud and security recommendations," https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/linux-threat-report-2021-1h-linux-threats-in-the-cloud-and-security-recommendations, 2021.

[9] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2011.

[10] Z. Mi, D. Li, H. Chen, B. Zang, and H. Guan, "(mostly) exitless VM protection from untrusted hypervisor through disaggregated nested virtualization," in *Proc. USENIX Security Symp.*, 2020.

[11] The Linux kernel development community, "Page migration — the linux kernel documentation," https://www.kernel.org/doc/html/v5.4/vm/page_migration.html, 2023.

[12] ——, "Transparent hugepage support — the linux kernel documentation," https://www.kernel.org/doc/html/v5.4/vm/transhuge.html, 2023.

[13] Intel, "Find the Size of Enclave Page Cache (EPC)," https://www.intel.com/content/www/us/en/support/articles/000089550/software/intel-security-products.html, 2022.

[14] C. Castes, A. Ghosn, N. S. Kalani, Y. Qian, M. Kogias, M. Payer, and E. Bugnion, "Creating trust by abolishing hierarchies," in *Proc. Wrkshp. Hot Topics in Operating Systems (HotOS)*, 2023.

[15] D. Magenheimer, "In-kernel memory compression," https://lwn.net/Articles/545244/, 2013.

[16] C. Li, "Blindfold," https://github.com/caihuali95/blindfold/, 2024.

[17] C. Li, S.-s. Lee, M. H. Yun, and L. Zhong, "MProtect: Operating system memory management without access," *arXiv preprint arXiv:2212.12671*, 2022.

[18] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2015.

[19] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A lightweight secure kernel-level execution environment for ARM," in *Proc. of Network and Distributed System Security Symp. (NDSS)*, 2016.

[20] M. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *Proc. of Network and Distributed System Security Symp. (NDSS)*, 2019.

[21] The Linux kernel development community, "Virtual memory primer — the linux kernel documentation," https://docs.kernel.org/admin-guide/mm/concepts.html#id1, 2023.

[22] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: Arming TrustZone with user-space enclaves." in *Proc. of Network and Distributed System Security Symp. (NDSS)*, 2019.

[23] M. Jones, “User space memory access from the linux kernel,” https://developer.ibm.com/articles/l-kernel-memory-access/, 2010.
[24] B. Dooks and K. Hu, “Investigating kernel user-space access,” https://www.codethink.co.uk/articles/2020/investigating-kernel-user-space-access/, 2020.
[25] FreeBSD, “Freebsd kernel developer’s manual, copy,” https://www.freebsd.org/cgi/man.cgi?query=copyin&sektion=9, 2020.
[26] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’keeffe, M. L. Stillwell *et al.*, “SCONE: Secure linux containers with Intel SGX,” in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2016.
[27] J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Transactions on Computer Systems (TOCS)*, 1984.
[28] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Trans. Information and System Security (TISSEC)*, 2012.
[29] S. Checkoway and H. Shacham, “Iago attacks: Why the system call API is a bad untrusted RPC interface,” in *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.
[30] ARM, “ARM trusted firmware,” https://www.trustedfirmware.org/, 2023.
[31] RustCrypto, “Rustcrypto project,” https://github.com/RustCrypto, 2023.
[32] LMbench - Tools for Performance Analysis, http://lmbench.sourceforge.net, 2013.
[33] OpenCV, “Example code of using DNN module in OpenCV,” https://docs.opencv.org/3.4.15/d5/de7/tutorial_dnn_googlenet.html, 2021.
[34] A. Talvensaari, “Unnecessary gettimeoftheday / ustime calls or mstime caching,” https://github.com/redis/redis/issues/2552, 2015.
[35] guybe7, “Excessive calls to mstime(),” https://github.com/redis/redis/issues/7053, 2020.
[36] LTP - Linux Test Project, https://linux-test-project.github.io/, 2022.
[37] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *Proc. Int. Wrkshp. Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.
[38] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, “Intel® software guard extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave,” in *Proc. Int. Wrkshp. Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.
[39] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr, “Exokernel: An operating system architecture for application-level resource management,” in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 1995.
[40] ARM, “Introducing Arm confidential compute architecture,” https://developer.arm.com/documentation/den0125/0100/Arm-CCA-Software-Architecture, 2021.
[41] Intel, “Intel trust domain extensions (Intel TDX),” https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2023.
[42] Assured Information Security, Inc., “Bareflank,” https://bareflank.github.io/hypervisor/, 2023.
[43] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: formal verification of an OS kernel,” in *Proc. ACM Symp. Operating Systems Principles (SOSP)*, 2009.
[44] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, “CertiKOS: An extensible architecture for building certified concurrent OS kernels,” in *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, 2016.
[45] A. Bhattacharyya, U. Tesic, and M. Payer, “Midas: Systematic kernel TOCTTOU protection,” in *Proc. USENIX Security Symp.*, 2022.
[46] I. Molnar, “A description of what robust futexes are,” https://docs.kernel.org/locking/robust-futexes.html.

Appendix

We conduct a case study of why and how the OS accesses user space using the Linux kernel. We note that others have studied user-space access by Linux kernel in the context of protecting the kernel from TOCTTOU attacks, e.g., most recently Midas [45]. The focus of that literature is on kernel vulnerabilities due to malicious users, not protecting users from an untrusted kernel.

A. Non-semantic access

The vast majority of kernel’s accesses to user space are concerned with the kernel moving user-space data. In these cases, the kernel does not need the semantics of the data being moved. We call such access non-semantic.

`read/write` syscalls. This pair of syscalls are extensively used for data exchange with I/O, including the filesystem. The kernel is simply responsible for copying data between memory region pointed by the `buf` argument and the file.

Demand paging. In the case of a page fault because the kernel loads file-backed pages on demand, the kernel calls the registered `fault` file operation to let the device driver prepare the frame, then updates the user page table to map the prepared physical frame to the user address space.

Swapping. When the number of free physical frames is low, the kernel may swap out some cold user pages from the physical memory to the external storage like disk and invalidate the corresponding user page table entries. When the user process accesses a swapped out page, it triggers a page fault. The kernel swaps in the page to the physical memory, remaps it to the user address space, and lets the process continue.

Memory compression [15]. Instead of swapping user pages to the external storage, the kernel can compress them after invalidating the associated page table entries. When next time the user process access a compressed page, the kernel decompresses the page and remaps it to the user address space.

Page migration. To reduce the latency of accessing memory in a NUMA system or to mitigate the problem of memory fragmentation, the kernel may migrate the content of a physical frame to another and update the associated user page tables accordingly. Linux provides two syscalls, `move_pages` and `migrate_pages`, to move user pages among nodes.

B. Semantic access

In some cases, the Linux kernel does need to understand the user-space data it accesses. We say such accesses are *semantic*. All semantic accesses share three properties. First, they are *well-defined* in spatial (where) and temporal (when) boundaries. Second, the user process knows when and where the kernel access its address space. Third, the kernel reads/writes the user space through well-defined interface, namely, `copy_to_user` and `copy_from_user`. These properties are the foundations to our solution of semantic access (See §IV-B).

Syscall argument passing. The most common cases are argument passing in syscalls. For example, the kernel must understand `pathname` in an `open` syscall to open the file. In these cases, the kernel accesses the user space *during* the syscalls (when) and in the region defined by the arguments (where).

Most syscalls pass an argument in a single transaction. That is, the kernel copies the user data specified by the argument to the kernel space when it handles the syscall. The kernel does not keep accessing the user data afterwards. Prior works leverage this and copy the argument data into a buffer managed by the TCB at call time for the kernel to access later.

The `futex` syscall, however, is a notable exception that does not work with the buffer-based approach. When the futex syscall is invoked, the kernel syscall handler reads the user-space word specified by the `futex` in order to determine if and how the in-kernel wait queue for the `futex` should be updated. The kernel considers reading the `futex` and updating its wait queue as a critical section that must be done atomically, because other threads may update the `futex` concurrently.

Robust `futex`. When a thread terminates unexpectedly while holding a `futex`, other threads waiting for the `futex` may end up waiting forever. Robust `futex` [46] solves this problem with a collaboration between glibc and the kernel. glibc creates and manages a list of all `futex`es held by the thread. The thread uses the syscall `set_robust_list` to inform the kernel where the head of the list is. When the thread terminates, the kernel traverses this list starting from its head.

`clone` syscall. When a user process makes a `clone` syscall to create a new process or a new thread, it can set the `CLONE_CHILD_CLEARTID` flag (`flags`) and pass the address of the child thread identifier (`child_tid`) to the kernel. When the child exits, the kernel will clear the child thread identifier by writing to the address specified by `child_tid`. Another syscall `set_tid_address` is similar.

Call stack write. In two cases, the kernel must write to the user-space memory defined by the call stack. First, when creating a process from a binary, the kernel must prepare its call stack by placing arguments at the bottom of the stack. Second, during signal handling, the kernel prepares the call stack (or signal stack) before handing control to a user-space signal handler. The signal handler can either use the call stack or an alternate *signal stack* (also in the user space), defined by the `sigaltstack` syscall when the signal handler is being established using the `sigaction` syscall. In both cases, the semantic write is well-defined in time and space.