Enabling Lightweight Privilege Separation in Applications with MicroGuards

Zahra Tarkhani[1] and Anil Madhavapeddy[2(✉)]

[1] Microsoft Research Cambridge, Cambridge, UK
[2] University of Cambridge, Cambridge, UK
ztarkhani@microsoft.com

Abstract. Application compartmentalization and privilege separation are our primary weapons against ever-increasing security threats and privacy concerns on connected devices. Despite significant progress, it is still challenging to privilege separate inside an application address space and in multithreaded environments, particularly on resource-constrained and mobile devices. We propose MicroGuards, a lightweight kernel modification and set of security primitives and APIs aimed at flexible and fine-grained in-process memory protection and privilege separation in multithreaded applications. MicroGuards take advantage of hardware support in modern CPUs and are high-level enough to be adaptable to various architectures. This paper focuses on enabling MicroGuards on embedded and mobile devices running Linux kernel and utilizes tagged memory support to achieve good performance. Our evaluation show that MicroGuards add small runtime overhead (less than 3.5%), minimal memory footprint, and are practical to get integrated with existing applications to enable fine-grained privilege separation.

1 Introduction

More than ever, we depend on highly connected computing systems in today's world, where over 6.3 Billion people use smartphones, and 35.82 billion IoT (Internet of Things) devices are installed worldwide [49]. Our growing reliance on edge-cloud services in recent years has been constantly and increasingly threatened by a wide range of security and privacy breaches at scales never seen before [4,5,41,53]. The attack surface of modern applications includes a mixture of traditional attack vectors with new threats within/across various dependencies and system abstractions.

Many software attacks target sensitive content in an application's address space, usually through remote exploits, malicious third-party libraries, or unsafe language vulnerabilities. Processing highly sensitive data in a single large compartment (e.g., process or enclave) leads to real threats that require effective

This paper will appear at the ACNS-SecMT2023 (Security in Mobile Technologies).
Z. Tarkhani—This work was done when the author was affiliated with the University of Cambridge.

J. Zhou et al. (Eds.): ACNS 2023 Workshops, LNCS 13907, pp. 571–598, 2023.
https://doi.org/10.1007/978-3-031-41181-6_31

protection against: *(i)* attackers can exploit vulnerabilities in less secure parts of the code to leak information, escalate privileges, or take control of the application or even the host. *(ii)* an application's secret data (e.g., private keys or user passwords) can be leaked in the presence of untrusted code parts or compromised third-party libraries like OpenSSL [23]; *(iii)* privileged functions or modules can be misused to access private content [22]; *(iv)* applications written in memory-safe languages such as Rust or OCaml are vulnerable via unsafe external libraries that jeopardize all other safety guarantees [6,35]; and *(v)* in multithreaded use cases, attackers can exploit vulnerabilities (e.g., TOCTOU or buffer overflows) so the compromised thread can access sensitive data owned by other threads [1]. This whole class of attacks could be avoided by providing a practical way to enforce the least privilege within a shared address space. Table 1 summarizes some of these real threats that intra-process protection is effective against.

Hence, the importance of in-address space security threats results in significant improvement in hardware support for efficient memory isolation [9,11,31,58]. However, existing simple APIs for utilizing such hardware features are not effective due to the complexity of attacks as well as various hardware limitations [20,43,55] in security and performance particularly for resource constrained devices. These systems mainly require specific programming languages or rely on x86 features which are not practical for wide range of IoT and mobile devices.

Table 1. A representative selection of vulnerabilities that cause sensitive content leakage. The attacks with a tick can be mitigated by using **MicroGuards** protection.

	example CVE	Description	MicroGuards
In-Process threats	CVE-2021-3450	Improper access control in shared library	✓
	CVE-2021-29922	unsafe language binding	✓
	CVE-2021-31162	Rust runtime memory corruption	✓
	CVE-2019-9345	Shared mapping bug	✓
	CVE-2021-45046	thread-based privilege escalation	✓
	CVE-2019-9423	missing bounds check	✓
	CVE-2019-15295	unsafe third party library	✓
	CVE-2019-1278	unsafe third party library	✓
	CVE-2018-0487	unsafe third party library	✓
	CVE-2017-1000376	unsafe native bindings	✓
	CVE-2014-0160	Heartbleed bug	✓
	CVE-2021-3177	Python ctypes memory leak	✓
	CVE-2021-28363	Python ctypes memory leak	✓
Other	CVE-2018-0497	SW side-channels	
	CVE-2017-5754	HW side-channels	

Many security-sensitive applications such as OpenSSH [44] rely on process-based isolation to separate their components into different privileged processes. However, this usually requires redesigning an application from scratch using a

multiprocess architecture (e.g., Chrome) and is difficult for many multithreaded applications such as web servers. Previous work such as Privtrans [18] and Wedge [16] provide automatic process-based isolation of applications with a huge overhead ($\approx 80\% - 40x$ slowdown).

Conventional process abstractions such as `fork` introduce security and efficiency issues [13], and alternatives such as `clone` are not fine-grained enough to switch between data sharing and copying between process address spaces for security-critical resources. This lack of flexibility in the underlying interfaces means developers cannot easily prevent in-process attacks, and so multithreaded applications are difficult to privilege separate. This class of attacks could be avoided by providing a practical way to protect memory within an address space.

In this paper, we present MicroGuards, a new OS abstraction for enforcing least privilege on slabs of memory within the same address space. It takes advantage of modern hardware features to provide a flexible and efficient way to define trust boundaries to isolate sensitive data while supporting familiar APIs for secure multithreading and memory management. We provide a virtual memory tagging and access control abstraction within the kernel, then extend the kernel to support mapping MicroGuards to threads; hence, any thread can selectively protect or share its memory compartments from untrusted code within itself or from any untrusted thread (see Fig. 1).

Hence, we designed a new memory compartmentalisation abstraction to overcome this limitation efficiently. MicroGuards virtual memory tagging layer bypasses most of the kernel's paging abstraction to enable isolated blocks of tagged memory which could be mapped to the undelying hardware features such as ARM MD (memory domains) or MTE (memory tagged extension) for stronger isolation enforcement and performance optimization. Moreover, these hardware features are difficult to use securely (require a strong access control mechanism) and portably due to differing semantics across the Linux Kernel virtual memory abstraction and hardware provided features (Sect. 2). Note that MicroGuards virtual memory layer can also be enabled with available simple address space translation mechanism and without hardware-based memory tagging capabilities. However, it is specifically designed for properly utilizing such beneficial hardware security features. Hence, MicroGuards is a high-level OS abstraction that aims to:

- develop a new kernel-assisted mechanism based on mutual-distrust for intra-process privilege separation that supports isolating private contents, a secure multithreading model, and secure communication within a shared address space.
- explain how to utilize modern CPU facilities for efficient memory tagging to avoid the overhead of existing solutions (due to TLB flushes, per-thread page tables, or nested page table management).
- show that the implementation is sufficiently lightweight ($\approx 5K$ LoC) to be practical for IoT and mobile devices with a minimal memory footprint.

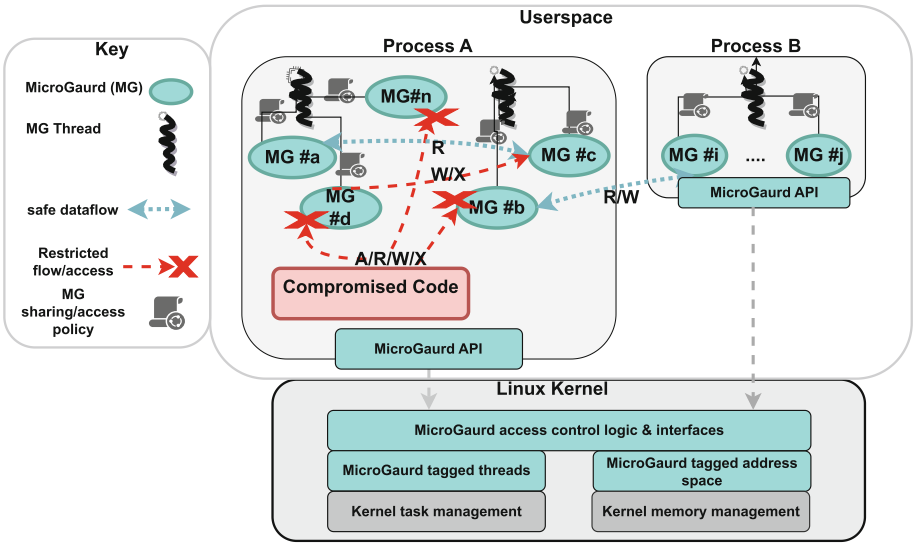

Fig. 1. High-level architecture of MicroGuards: it provides in-process isolation as well as thread-granularity privilege separation so each MicroGuard thread can tag itself, its address space, and define its own trust boundaries.

– evaluate our implementation using real-world software such as Apache HTTP server, OpenSSL, and Google's LevelDB, which shows MicroGuards add negligible runtime overhead for lightly modified applications.

The remainder of this paper elaborates on the CPU hardware features we use (Sect. 2), describes the architecture (Sect. 3) and implementation of MicroGuards (Sect. 4), presents an evaluation (Sect. 5) and the tradeoffs of our approach (Sect. 6).

2 Background

2.1 ARM VMSA

ARM virtual memory system architecture (VMSA) is tightly integrated with the security extensions, the multiprocessing extensions, the Large Physical Address Extension (LPAE), and the virtualization extensions. VMSA provides MMUs that control address translation, access permissions, and memory attribute determination and checking for memory accesses. The extended VMSAv7/v8 provides multiple stages of memory system control; for operation in Secure state (e.g., EL1&0 stage 1 MMU) and for operation in Non-secure state (e.g., EL2 stage 1 MMU, EL1&0 stage 1 MMU, and EL1&0 stage 2 MMU). VMSAv8.5 adds more MMUs for additional isolation in the secure world. Each MMU uses a set of address translations and associated memory properties held in TLBs. If

an implementation does not include the security extensions, it has only a single security state, with a single MMU with controls equivalent to the Secure state MMU controls. A similar argument is valid for when am implementation does not include the virtualization extensions.

System Control coprocessor (CP15) registers control the VMSA, including defining the location of the translation tables. They include registers that contain memory fault status and address information. The MMU supports memory accesses based on memory sections or pages, supersections consist of 16MB blocks of memory, sections consist of 1MB blocks of memory or 64 KB blocks of memory, and pages consist of 4 KB blocks of memory. Operation of MMUs can be split between two sets of translation tables, defined by the Secure and Non-secure copies of `TTBR0` and `TTBR1`, and controlled by `TTBCR`. For `hyp` mode stage 1, The `HTTBR` defines the translation table for EL2 MMU, controlled by `HTCR`. For stage 2 translation, The `VTTBR` defines the translation table, controlled by `VTCR`. Access to a memory region is controlled by the access permission bits and the domain field in the TLB entry.

ARM Memory Domains (MDs). A domain is a collection of contiguous memory regions. The ARM VMSAv7 architecture supports 16 domains, and each VMSA memory region is assigned to a domain. First-level translation table entries for page tables and sections include a domain field. Translation table entries for super-sections do not include a domain field (super-sections are defined as being in domain 0). Second-level translation table entries inherit a domain setting from the parent first-level page table entry. Each TLB entry includes a domain field. A domain field specifies which domain the entry is in, and a two-bit field controls access to each domain in the Domain Access Control Register (`DACR`). Each field enables access to an entire domain to be enabled and disabled very quickly without TLB flushes so that whole memory areas can be swapped in and out of virtual memory very efficiently. Hence `DACR` controls the behavior of each domain and is not guarded by the access permissions for TLB entries in that domain. Also, `DACR` defines the access permission for each of the sixteen isolation domains. The `DACR` is a 32-bit read/write register and is accessible only in privileged modes. When the security extensions are implemented `DACR` is a banked register, and write access to the secure copy of the register is disabled when the `CP15SDISABLE` signal is asserted high. To access the `DACR` you read or write the `CP15` registers. For example: '`MRC` $p15, 0, <Rt>, c3, c0, 0$' for reading from `DACR` and '`MCR` $p15, 0, <Rt>, c3, c0, 0$' for writing to `DACR`. Data Fault Status Register (DFSR) holds status information about the last data fault in MDs. It is a 32-bit read/write register, accessible only in privileged modes. These registers are banked when security extensions are enabled, so we could have separate 16 domains inside TrustZone secure world as well as the normal world.

Table 2. ARM memory domains access permissions

Mode	Bits	Description
No Access	00	Any access causes a domain fault
Manager	11	Full accesses with no permissions check
Client	01	Accesses are checked against the page tables
Reserved	10	Unknown behaviour

The four possible access rights for a domain are No Access, Manager, Client, and Reserved (see Table 2). Those fields let the processor *(i)* prohibit access to the domain mapped memory–No Access; *(ii)* allow unlimited access to the memory despite permission bits in the page table– Manager; or *(iii)* let the access right be the same as the page table permissions–Client. Any access violation causes a domain fault, and changes to the DACR are low cost and activated without affecting the TLB.

ARM MDs look like a good building block for in-process memory protection. Changing domain permissions does not require TLB flushes, and they do not require extensive modifications to the kernel memory management structures that might otherwise introduce security holes due to inevitable TLB and memory management bugs [61].

Though ARM MDs are a useful isolation primitive in concept, the current hardware implementation and OS support suffer from significant problems that have prevented their broader adoption:

Scalability: ARM relies on a 32-bit `DACR` register and so supports only up to 16 domains. Allocating a larger register (e.g., 512 bits) would mean larger page table entries or additional storage for domain IDs.

Flexibility: Unlike Intel MPK, ARM-MDs only apply to first-level entries; the second-level entries inherit the same permissions. This prevents arbitrary granularity of memory protections to small page boundaries and reduces the performance of some applications [21]. Also, the `DACR` access control options do not directly mark a domain as read-only, write-only, or exec-only. So the higher-level VM abstraction should resolve these issues.

Performance: Changing the `DACR` is a fast but privileged operation, so any change of domain access permissions from userspace require a system call. This is unlike Intel MPK that makes its Protection Key Rights Register (PKRU) accessible directly from userspace.

Userspace: There is no Linux userspace interface for using ARM-MD; it is only used within the kernel to map the kernel and userspace into separate domains. In contrast, Linux already provides some basic support for utilizing Intel MPK from userspace.

Security: Though the `DACR` is only accessible in privileged mode, any syscall that changes this register is a potential breach that could cause the attacker to gain

full control of the host kernel (e.g., through the misuse of the `put_user/get_user` kernel API in CVE-2013-6282). Also, since only 16 domains are supported, guessing other domains' identifiers is trivial, making it essential not to expose these directly to application code.

Address Space Identifier. The VMSA permits TLBs to hold any translation table entry that does not directly cause a translation fault or an access flag fault. To reduce the software overhead of TLB maintenance, the VMSA differentiates between *global pages* and *process-specific pages* through the Address Space Identifier (ASID). A global virtual memory page is available for all processes on the system, and a single cache entry can exist for this page translation in the TLB. A non-global virtual memory page is process-specific, associated with a specific ASID. The ASID identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without maintaining the TLB structures. Hence, multiple TLB entries can exist for the same page translation, but only TLB entries that are associated with the current ASID are available to the CPU (x86 supports a similar mechanism, called PCID). On ARMv7, the current ASID is defined by the Context ID Register (CONTEXTIDR), and on ARMv8, the ASID is defined by the translation table base registers that causes better performance compare to ARMv7. Each `TTBR` contains an ASID field, and the `TTBCR.A1` field selects which ASID to use. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits. ASIDs/PCIDs are useful for relatively faster context switching [38] and more efficient page table isolation as shown in design of kernel page-table isolation (KPTI or PTI, previously called KAISER [27]) for mitigating Meltdown vulnerability [37].

MTE and PAC. Memory Tagging Extension (MTE), also called memory coloring, is introduced in Armv8.5-A. Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory (this is the Tag Granule). Tagging memory implements the lock. Hence, pointers and virtual addresses are modified to contain the key. In order to implement the key bits without requiring larger pointers, MTE uses the TBI (top byte ignore) feature of the Armv8-A Architecture. When TBI is enabled, the top byte of a virtual address is ignored when using it as an input for address translation similar to PAC (Pointer Authentication Code) design. This allows the top byte to store metadata. Memory tagging and pointer authentication both use the upper bits of an address to store additional information about the pointer: a tag for memory tagging, and a PAC for pointer authentication. Both technologies can be enabled at the same time. The size of the PAC is variable, depending on the size of the virtual address space. When memory tagging is enabled at the same time, there are fewer bits available for the PAC.

MTE adds a new memory type, Normal Tagged Memory, to the Arm Architecture. A mismatch between the tag in the address and the tag in memory can be configured to cause a synchronous exception or to be asynchronously

reported. When the asynchronous mode is enabled, upon fault, the PE updates the `TFSR_EL1` register. Then the kernel detects the change during context switching, return to `EL0`, kernel entry from `EL1`, or kernel exit to `EL1`. MTE is currently supported by LLVM, and when it is enabled, a call to `malloc()` will allocate the memory and assign a tag for the buffer. The returned pointer will include the allocated tag. If software using the pointer goes beyond the limits of the buffer, the tag comparison check will fail. This failure will allow us to detect the overrun. Similarly, for use-after-free, on the call to `malloc()` the buffer gets allocated in memory and assigned a tag value. The pointer that is returned by `malloc()` includes this tag. The C library might change the tag when the memory is released. If the software continues to use the old pointer, it will have the old tag value, and the tag-checking process will catch it.

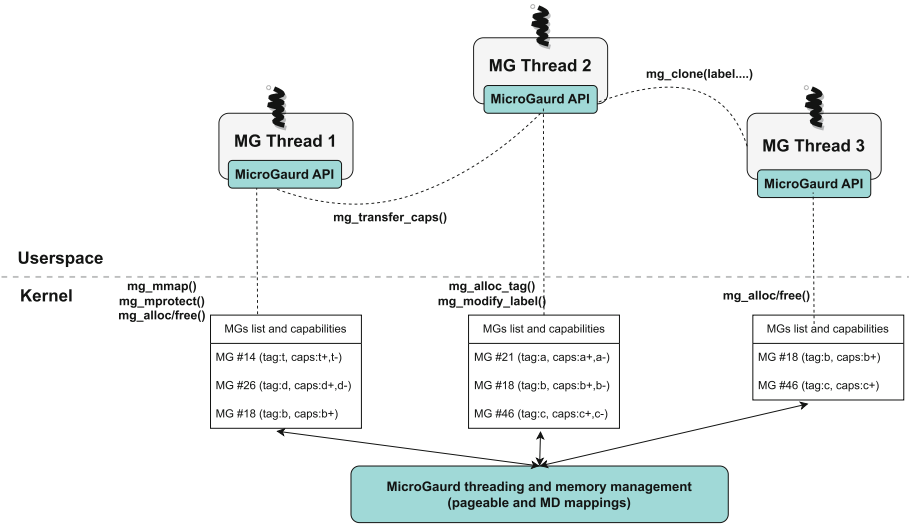

Fig. 2. Simple MicroGuards simple threading example: each MicroGuard thread is a security principal, it can define security policies for controlling its own MicroGuards collection, and pass its capabilities to other threads for secure sharing. The kernel then enforces MicroGuard security policies and handles its virtual memory management.

3 MicroGuards

We now describe the implementation of MicroGuards, which is an abstraction over the underlying kernel and hardware memory management for efficient intra-process isolation. MicroGuards abstraction has an emphasis on security, performance, and extensibility to support various hardware memory tagging primitives through a higher-level interface that hides the hardware limitations (Sect. 2.1).

3.1 Design Principles

The MicroGuards interface aims to enforce least privilege principle for memory accesses via the following guidelines:

Fine-Grained Strong Isolation: All threads of execution should be able to define their security policies and trust models to selectively protect their sensitive resources. Current OS security models of sharing ("everything-or-nothing") are not flexible enough for defining fine-grained trust boundaries within processes or threads (lightweight processes).

Performance: Launching MicroGuards, changing their access permissions, sharing across processes, and communications through capability passing should have minimal overhead. Moreover, untrusted (i.e., MicroGuards-independent) parts of applications should not suffer any overhead.

Efficiency: MicroGuards should be lightweight enough even for mobile and IoT devices running on a few megabytes of memory and slow ARM CPUs.

Compatibility: It is difficult to provide strong security guarantees with no code modifications, and MicroGuards is no exception. We move most of these modifications into the Linux kernel (increasingly popular for embedded deployments [2]) and provide simple userspace interfaces. MicroGuards should be implemented without extensive changes to the Linux and not depend on a specific programming language, so existing applications can be ported easily.

To achieve fine-grained isolation with mutual-distrust, we need a security model that lets each thread protect its own MicroGuards from untrusted parts of the same thread as well as other threads and processes. Simply providing POSIX memory management (e.g. `malloc` or `mprotect`) is inadequate. As a simple example, attackers can misuse the API for changing the memory layout of other threads MicroGuards or unauthorized memory allocation. The MicroGuards interface needs to *(i)* provide isolation within a single thread; *(ii)* be flexible for sharing and using MicroGuards between threads, and *(iii)* provides the capability to restrict unauthorized permission changes or memory mappings modification of allocated MicroGuards. Previous work such as ERIM [55] or libMPK [43] does not offer such security guarantees since their focus is more on performance and domain virtualization.

We derive inspiration from Decentralized Information Flow Control (DIFC) [34] but with a more constrained interface – by not supporting information flow within a program, we avoid the complexities and performance overheads that typically involves. Existing DIFC kernels such as HiStar [59] achieve our isolation goals, but requires a non-POSIX-based OS that opposes our compatibility goal. To have a practical and lightweight solution, we therefore built MicroGuards over a modified Linux kernel, and internally utilizing modern hardware facilities such as ARM MDs for good performance.

3.2 Threat Model and Assumptions

This paper focuses on two types of threats. First, memory-corruption based threats inside a shared address space that lead to sensitive information leakage; these threats can be caused by bugs or malicious third-party libraries (see Table 1). Second, attacks from threads that could get compromised by exploiting logical bugs or vulnerabilities (e.g., buffer overflow attacks, code injection, or ROP attacks). We assume the attacker can control a thread in a vulnerable multithreaded application, allocate memory, and fork more threads up to resource limits by the OS and hardware. The attacker will try to escalate privileges through the attacker-controlled threads or gain control of another thread, e.g., by manipulating another thread's data or via code injection. The adversary may also bypass protection regions by exploiting race conditions between threads or by leveraging confused-deputy attacks.

MicroGuards thus provides isolation in two stages: firstly within a single thread (through mg_lock/unlock calls), and then across threads in the same process. We consider threads to be security principals that can define their security policies based on mutual-distrust within the shared address space. We protect each thread's MicroGuards against unauthorized, accidental, and malicious access or disclosure. Therefore, the TCB consists of the OS kernel, which performs this enforcement. It also assumes developers correctly specify their policies through the userspace interface for allocating MicroGuards and transferring capabilities.

MicroGuards are not protected against covert channels based on shared hardware resources (e.g., a cache). Systems such as Nickel [47] or hardware-assisted platforms such as Hyperflow [25] could be a helpful future addition for side-channel protection on MicroGuards.

3.3 MicroGuards Access Control Mechanism

Each MicroGuard is a contiguous allocation of memory that (by default) only its owner thread can access, add/remove pages to/from it, and change its access permission. Our modified Linux kernel enforces the access control via a dynamic security policy based on DIFC [59] and a simpler version of the Flume [34] labeling model.

Each MicroGuard thread t has one label L_t that is the set of its unique tags. Privileges are represented in forms of two capabilities θ^+ and θ^- per tag θ for adding or removing tags to/from labels. These capabilities are stored in a capability list C_p per thread p. To improve its performance, MicroGuards have only one unique secrecy tag assigned internally by the kernel when created by `mg_create`. For improving security, none of MicroGuards API propagates tags in the userspace; all APIs access control is done internally within the kernel. The kernel allows information flow from α to β only if $L_\alpha \subseteq L_\beta$. Every thread p may change its label from L_i to L_j if it has the capability to add tags present in L_j but not in L_i, and can drop the tags that are in L_i but not in L_j. This is formally declared as $(L_j - L_i \subseteq C_p^+) \wedge (L_i - L_j \subseteq C_p^-)$.

Table 3. MicroGuards access control system calls. Pi represents principal i, L as a label that is a list of tags ($t*$) and their capabilities ($c*$).

syscalls	Description
mg_alloc_tag()$\rightarrow t$	allocate a unique tag
mg_modify_label(L)	modify a thread's label/tag
mg_transfer_caps($L \rightarrow c*, p$)	passing capabilities to thread p
mg_declassify($L \rightarrow t*$)	thread declassification or endorsement
mg_grant($L \rightarrow t*, p1, p2$)	adds an acts-for or a delegation link
mg_revoke_grant($L \rightarrow t*, p1, p2$)	removes an acts-for or a delegation link
mg_lock ($L \rightarrow t*$)	disables access to an object
mg_unlock ($L \rightarrow t*$)	enables access to a locked object
mg_clone ($L, int(*fn)(void*)...) \rightarrow p$	creates a thread

When a thread has θ^+ capability for MicroGuard θ, it gains the privilege to only access MicroGuard θ with the permission set by its owner (read/write/execute). The access privileges to each MicroGuard can be different; hence, two threads can share a MicroGuard, but the access privileges can differ.

Having a θ^- capability lets it declassify MicroGuard θ. This allows the thread to modify the MicroGuard memory layout by add/remove pages to it, change permissions, or copy the content to untrusted sources. Unsafe operations like declassification require the thread to be an owner or an authority (`acts-for` relationship) then via `mg_grant` and `mg_revoke` calls (see Table 3).

3.4 MicroGuards Threads

Each MicroGuard thread may have multiple MicroGuards attached to it. There is no concept of inheriting capabilities by default (e.g., in the style of `fork`) as this makes reasoning about security difficult [13]. Here, a tagged thread can create a child by calling `mg_clone`; the child thread does not inherit any of its parent's capabilities. However, the parent can create a child with a list of its MicroGuards and selected capabilities as an argument of `mg_clone`. For instance, in Fig. 2, thread 3 is a child of MicroGuard thread 2, which only gets "plus" capabilities for both shared MicroGuards 18 and 46 via `mg_clone` with a specific Label passed by its parent thread.

For a MicroGuard to propagate, it must be through transferring capabilities; this can be done directly by calling `mg_transfer_caps` for "plus" capabilities and `mg_grant` for declassification. Both these operations are also possible via specific arguments of `mg_clone` syscall when creating a child thread. Figure 2 shows how each thread can use the MicroGuards API for creating tags, changing labels, and passing capabilities to other threads. For instance, thread 1 gains access to MicroGuard 18 by directly getting the b^+ capability from thread 2. Since it does not have the b^- capability, it cannot change MicroGuard 18 permissions or its memory mappings.

Table 4. Some of userspace MicroGuards memory management API. Each MicroGuard has an *id* and is a tagged kernel object internally. MicroGuards access control is checked within the kernel.

Name	Description
mg_create → id	Create a new MicroGuard
mg_kill(*id*)	Destroy a MicroGuard
mg_malloc(*id*, *size*) → void*	Allocate memory within a MicroGuard
mg_free(*id*, *void*∗)	free memory from a MicroGuard
mg_mprotect(*id*, ...)	change an MicroGuard's pages permission
mg_mmap(*id*, ...)→ void*	Map a page group to a MicroGuard
mg_munmap(*id*, ...)	Unmap all pages of a MicroGuard
mg_get(*id*)→ *perms*	Get a MicroGuard permission

Table 3 describes the userspace MicroGuard API. A thread can create a tag by calling `mg_alloc_tag`, and the kernel will create and return a fresh unique tag. The thread that allocates a tag becomes its owner and can give the capabilities for the new tag to other threads. Each thread specifies its security policies by mutating its labels via `mg_modify_label`, and can declassify its own MicroGuards via `mg_declassify`.

Threads can lock access or permission changes of their MicroGuards via `mg_lock`, which temporarily change MicroGuard tag to restrict any modifications of MicroGuards state. A locked MicroGuard can only be accessed by calling `mg_unlock`.

MicroGuards Memory Management. To provide in-process isolation with good performance (Sect. 3.1) we provide a virtual memory management abstraction within the kernel for MicroGuards-aware memory tagging, mappings, protection, page faults handling, and least privilege enforcement. This abstraction bypasses most of the kernel paging abstraction that improves its performance. Furthermore, it hides the intricacies of hardware domains. Then we provide a userspace library on top of our modified kernel, using our MicroGuards-specific system calls, for managing MicroGuards memory. An application creates a new MicroGuard by calling `mg_create`; the kernel creates a unique tag with both capabilities (since it is the owner) and adds it to the thread's label and capability lists, and returns a unique ID. A MicroGuard can be kernel-backed (just depending on commodity pagetable for isolation) or hardware-backed which maps a MicroGuard to finer-grained memory safety/tagging features. We extend the kernel VM layer to support MicroGuards and maintain a private per-MicroGuard virtual page table (`pgd_t`) that is loaded into the `TTBR` register when the thread needs to do memory operations inside an MicroGuard during a lightweight context switch. An internal MicroGuard data structure maintains its address space range and permissions as shown in the following codelisting 1.1.

```
struct mg_struct {
    //operation bitmaps: set to 1 if mg[i] is allowed to do this operation, 0 OW
    DECLARE_BITMAP(mg_Read, MG_MAX);
    DECLARE_BITMAP(mg_Write, MG_MAX);
    DECLARE_BITMAP(mg_Execute, MG_MAX);
    DECLARE_BITMAP(mg_Allocate, MG_MAX);
    int mg_id;
    struct mutex mg_mutex;
    struct mem_segment *mg_range;
   };
```

Listing 1.1. Internal MicroGuard data structure

Threads (or Linux tasks) in a process share the same `mm_struct` that describes the process address space. Having separate `mm_struct` for threads would significantly impact system performance, as all the memory operations related to page tables should maintain strict consistency [29]. Instead, we extend `mm_struct` to embed MicroGuard metadata within it as lightweight protected regions in the same address space as shown in Listing 1.2. It stores a per-MicroGuard `pgd_t` for threads and other metadata for memory management, fault handling, and synchronization.

The standard Linux kernel avoids reloading page tables during a context switch if two tasks belong to the same process. We modified `check_and_switch_context` to reload MicroGuard page tables and flush related TLB entries if one of the switching threads owns an MicroGuard. We further mitigate the flushing overhead using ASID tagged TLB feature and ARM MDs. We modify `mmap.c` to keep track of MicroGuard-mapped memory ranges and add `mg_mmap/mumap` operations.

The kernel `handle_mm_fault` handler is also extended to specially manage page faults in MicroGuard regions, so an MicroGuard privilege violation results in the handler killing the violating thread.

```
struct mm_struct {
...
#ifdef CONFIG_MG
    struct mg_struct *mg_metadata[MG_MAX];
    atomic_t num_mg;    /* number of mgs */
    pgd_t *mg_pgd_list[MG_MAX]; /*mg Page tables per threads.*/
    int curr_using_mg;
    spinlock_t sl_mg[MG_MAX];
    struct mutex mg_metadata_mut;
    DECLARE_BITMAP(mg_InUse, MG_MAX);
#endif
... };
```

Listing 1.2. Extending the Linux kernel mm_struct with MicroGuards metadata.

Example code 1.3 shows a basic way of using MicroGuards to protect sensitive content in a single thread. Then the owner thread maps pages to its MicroGuard by calling `mg_mmap` that updates the MicroGuard's metadata with its address space ranges. The kernel allows mappings based on the thread's labels and free hardware domains. If there is a free hardware domain, it maps pages to that domain and places it to MicroGuards cache. When the MicroGuards already

exists in the cache, further access to it is fast. When there is no free hardware domain, we have to evict one of the MicroGuards from the cache and map the new MicroGuard metadata to the freed hardware domain; this requires storing all the necessary information for restoring the evicted MicroGuard, such as its permission, address space range, and tag. The caching process can be optimized by tuning the eviction rate and suitable caching policies similar to libMPK [43].

The application uses `mg_malloc` and the MicroGuard ID to allocate memory within the MicroGuard boundaries (mg_malloc), and `mg_free` to deallocate memory or `mg_mprotect` to change its permissions (see Table 4). The owner thread can use `mg_lock` to restrict unauthorized access to it by accident or other malicious code; this is helpful for mitigating attacks inside a single thread. Then application developer can allow only his trusted functions or necessary parts of the code to gain access by calling `mg_unlock` (e.g., our single-threaded OpenSSL use case in Sect. 5.2).

```
/* create a microgaurd (i.e., mg_id) */
int mg_id = mg_create();

/* map a memory region to the mg */
memblock = (char*) mg_mmap(mg_id, addr, len, prot , 0, 0); //

// set permissions by mg_mprotect

/* allocate memory from mg */
private_blk = (char*) mg_malloc(mg_id, priv_len);

/* make mg inaccessible */
lock_mg(mg_id);

//... untrusted computations ....//

/* make mg accessible */
unlock_mg(mg_id);

//... trusted computations ....//

/* cleanup mg */
mg_free(private_blk);
mg_munmap(mg_id, memblock,len);
```

Listing 1.3. Basic MicroGuards usage

Our current implementation of MicroGuards utilizes ARM-MDs for efficient in-process virtual memory tagging; as a result, only code running in supervisor mode can change a domain's access control via the DACR register (Sect. 2.1) or remap private addresses to another domain through the TTBR domain bits. However, note that MicroGuards abstraction is designed to support similar hardware memory tagging features such as MTE and PAC with straightforward changes; mostly by replacing the backed for MicroGuards memory management API (`mg_malloc` layer) since the threading and other kernel changes are architecture-agnostic. Our API and mappings prevent unauthorized permission changes for MicroGuards, and we also do not provide a userspace API for direct modification of the DACR. Threads security policy enforcement is done by adding custom security hooks in the kernel's virtual memory management and

task handling layers. It checks access based on the correct flow of threads labels (Sect. 3.4). We extend the kernel page fault handler for MicroGuards-specific cases. Illegal access to MicroGuards causes domain faults which our handler logs (e.g., violating thread information) and terminates it with a signal.

4 Implementation

MicroGuards Kernel: The MicroGuards core access control enforcement and the security model is implemented in the form of a new Linux Security Module (LSM) [42] with only four custom hooks. The LSM initializes the required data structures, such as the label registry and includes the implementation of all access control system calls (Table 3) for enforcing least privilege. This includes locking MicroGuards, changing labels, transferring capabilities, authority operations, and declassification based on the labeling mechanism (Sect. 3.4).

We modify the Linux task structure to store the metadata required to distinguish MicroGuards tasks from regular ones. Specifically, we add fields for storing MicroGuards metadata, label/ownership as an array data structure holding its tags (each tag is a 32-bit identification whose upper 2 bits stores plus and minus capabilities), a capability list; all included as task credential data structure. We implemented a hash table-based registry to make operations (e.g., store, set, get, remove) on these data structures more efficient.

The LSM also provides custom security hooks for parsing userspace labels to the kernel (`copy_user_label`), labeling a task (`set_task_label`), checking whether the task is labeled (`is_task_labeled`), and checking if the information flow between two tasks is allowed (`check_labels_allowed`). These security hooks are added in various places within the kernel to MicroGuards are guarded against unauthorized access or permission change by either the POSIX API (e.g., mmap, mprotect, fork) or the MicroGuards API. For example, forking a labeled task should not copy its labels and capability lists, and this is enforced using the MicroGuards LSM hooks. As another example, to avoid a task performing unauthorized memory allocation into a random MicroGuard or mapping pages to it, the security hooks are in the kernel's virtual memory management layer where the MicroGuards memory management engine (Table 4) can enforce correct access.

The MicroGuards virtual memory abstraction is implemented as a set of kernel functions similar to their Linux equivalents (e.g., `do_mmap`, `do_munmap` and `do_mprotect`) with similar semantics but with additional arguments that are required for enforcing the least privilege on MicroGuards. When an application creates a MicroGuard by calling `mg_create` (or `mg_mmap` for the first time), a MicroGuard ID passed as an argument that is associated with in-kernel metadata, together with the MicroGuard tag, and its capabilities that would be added to the task credentials.

When MicroGuards are mapped to hardware domains, the exact physical domain number is hidden from the userspace code to avoid possible misuse of the API. The mappings between MicroGuards and hardware domains are maintained

through a cache-like structure similar to libmpk [43]. A MicroGuard is inside the cache if it is already associated with a hardware domain; otherwise, it evicts another MicroGuards based on the least recently used (LRU) caching policy while saving all require metadata for restoring the MicroGuard mapping and permission flags.

Users can get their MicroGuards permissions by calling `mg_get`, and quickly change its permission through `mg_mprotect` if the requested permission change matches one of the domain's supported options (Table 2) or undergo the small overhead of a dynamic security check otherwise. Any violation of MicroGuards permissions causes a MicroGuards fault that leads to the violating thread being terminated. To protect MicroGuards against API attacks, all memory management system calls check whether the caller thread has the appropriate capabilities using the security hooks.

Creating a MicroGuard adds a new tag and owner capabilities to the task credential, and the userspace library also provides a management API for modifying labels and capabilities. Each thread can use `mg_transfer_caps` for passing the plus capabilities to other threads, `mg_grant_revoke` for handling authorities, `mg_lock` to prohibit access to a MicroGuard, and `mg_unlock` to restore access. The `mg_lock/unlock` operations are helpful in limiting in-process buggy code from accessing MicroGuards content.

Userspace: To reduce the size of the TCB, we did not modify existing system libraries and instead provided a userspace library to invoke MicroGuards system calls. This library supports a familiar API for memory management within a MicroGuard, including `mg_malloc` and `mg_free` for memory management. We provide a custom memory allocator similar to HeapLayer [15] that allocates memory from an already mapped MicroGuard. For each MicroGuard, there is a memory domain metadata structure that keeps essential information such as the MicroGuard address space range (base and length) and the two lists of free blocks from the head and tails of the MicroGuard region that is used when searching for free memory.

5 Evaluation

We evaluated our implementation of MicroGuards on a Raspberry Pi 3 Model B [3] that uses a Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor with 32 KB L1 and 512 KB L2 cache memory, running a 32-bit unmodified Linux kernel version 4.19.42 and glibc version 2.28 as the baseline. We use microbenchmarks and modified applications to evaluate MicroGuards in terms of security, performance, and usability (Sect. 3.1 and Sect. 2.1) by answering the following questions:

– What is the initialization and runtime overhead of MicroGuards? How does using hardware domains impact performance?

– Are MicroGuards practical and adaptable for real-world applications? How much application change and programming effort is required? What is the performance impact? How does it perform in a multi-threaded environment?
– What is the memory footprint of MicroGuards? How much memory does it add (statically and dynamically) to both the kernel and userspace?

5.1 Microbenchmarks

Creating MicroGuards: Table 5 tests the cost of creating and mapping pages to MicroGuards using `mg_mmap` when MicroGuards are directly mapped to hardware domains, 1MB aligned memory regions with only 16 MicroGuards support, as compared to virtualized MicroGuards when there is no free hardware domain and requires evicting MicroGuards from the cache. The results show that the direct use of hardware domains improves MicroGuards performance by 4.9% compare to the virtualized one. Note that creating MicroGuards is usually a one-time operation at the initial phase of an application.

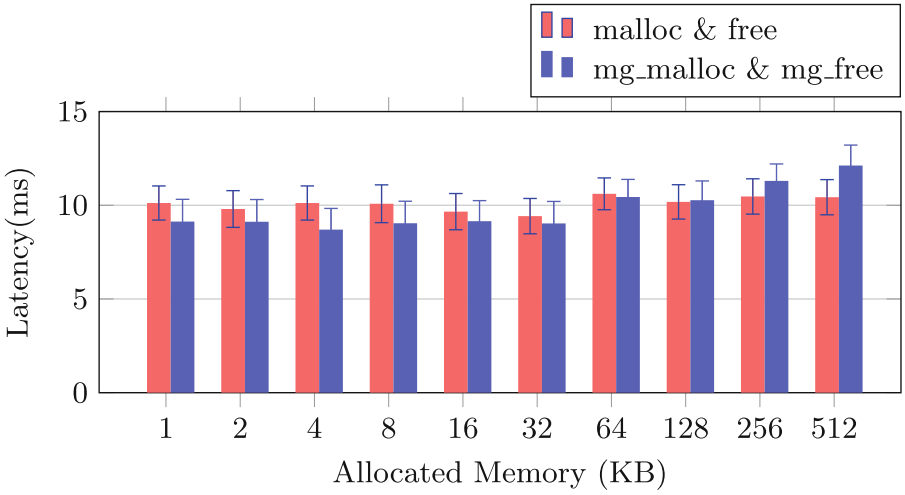

Fig. 3. Cost of MicroGuards memory allocation (malloc & free). On average `mg_malloc` outperforms `malloc` by a small rate (0.03%).

Table 5. Cost of creating MicroGuards when directly mapped to hardware domains vs virualised mapping that requires MicroGuards caching. The results are average of 10000 runs.

Operation	Overhead	stddev
Direct mg_mmap/munmap	4.8%	±0.17%
Virtualised mg_mmap/munmap	10.01%	±0.15%

Memory Protection and Allocation

We measure the cost of memory protection for baseline Linux where protection is per-process, and on MicroGuard threads where protection is per-thread and either implemented in software or hardware.

Table 5 shows the average results of 10000 runs of our microbenchmark comparing the cost of `mg_mprotect` with `mprotect` on baseline kernel. The results show `mg_mprotect` is 1.12x slower than `mprotect`, but the MD-backed `mg_mprotect` is 1.14x faster than baseline for some permissions (none and r/w) that supported by `DACR` register and do not need a TLB flush. Note that since hardware memory domains do not have flexible access control options, we cannot benefit from a control switch of domains using the DACR register for all possible permission flags such as the RO, WO, and EO variants.

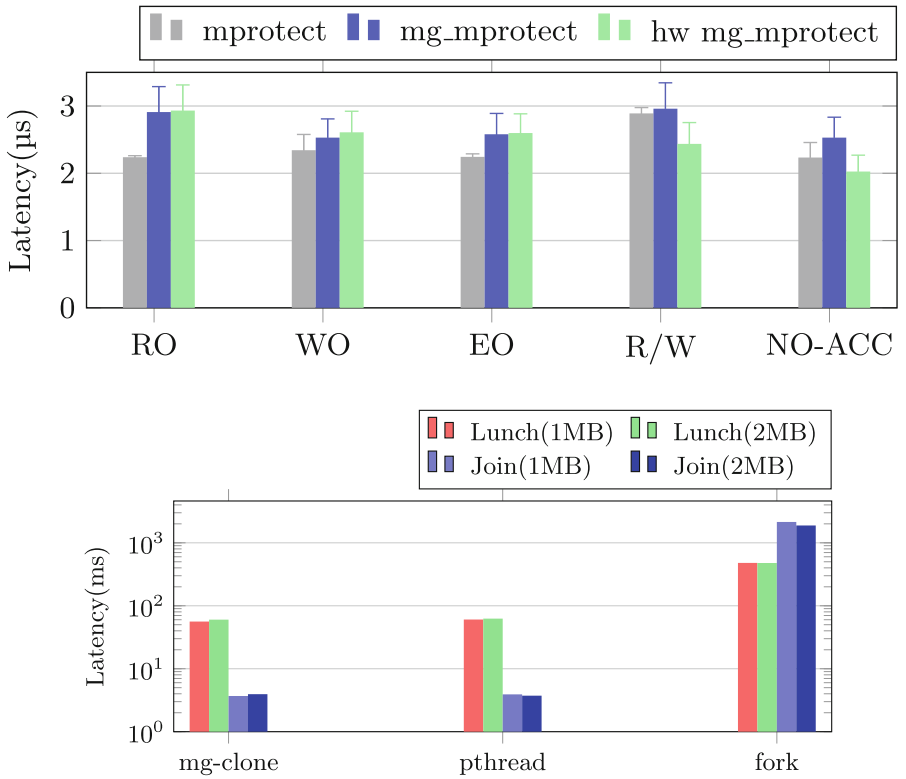

Fig. 4. Overhead of creating MicroGuard-enabled threads: the results are the average of 100000 runs with 1MB and 2MB heap sizes. On average, `mg_clone` latency is 5.39% lower than of `pthread_create`.

Table 6. Memory overhead of MicroGuards in Linux Kernel and userspace

Overhead	Linux Kernel	Userspace
Added LoC	3023	2405
Static Memory footprint	static(7 KB) slab(204 KB)	Static(10 KB)

Allocating memory using `mg_malloc` is on average 1.08x faster than glibc `malloc` for blocks ≤64 KB and introduces a small overhead (8.3%) for blocks greater than 64 KB (see Fig. 3). This cost can be optimised by using high-performance memory allocators. The results are average of running microbenchmarks 20000 times, and shows using MicroGuards provides reasonable overhead for memory allocation and permission changes.

Threading: We tested the cost of MicroGuard threading operations (creating and joining) through `mg_clone` that creates MicroGuard-aware threads. The test uses the `clone` syscall with minor modifications to restrict any credential sharing with the child by default (instead it provides additional clone options for passing parent's capabilities to its child). We implemented `mg_join` using `waitpid`. Figure 4 shows `mg_clone` outperforms `pthread_create` by 0.56% and `fork` by 83.01%. This gain is attributed to the MicroGuard operations simply doing less work for initializing new threads.

Codebase Overhead: Another factor towards the usability of MicroGuards is the size of the codebase, which is important both from a security perspective and the resource limitations of small devices. We implemented MicroGuards as a Linux kernel patch with no dependency on any userspace libraries. As Table 6 shows it adds less than $5.5K$ LoC in total to both the kernel ($\approx 3K$ LoC) and userspace ($2.5K$ LoC). It adds 7 KB to the kernel image size and adds 204 KB for kernel slabs at runtime. The userspace library only needs ≈10 KB of memory. These results show the MicroGuards memory footprint is small and suitable for many resource-constrained uses.

5.2 OpenSSL

Cryptographic libraries are responsible for securing all connected devices and network communication, yet have been a source or victim of severe vulnerabilities. Given these libraries' critical role, a single vulnerability can have a tremendous security impact. The well-known OpenSSL's Heartbleed vulnerability [23], for example, enabled attackers to access many servers' private data (up to 66% of all websites were vulnerable). More recently, GnuTLS suffered a significant vulnerability allowing anyone to passively decrypt traffic (CVE-2020-13777). Lazar et al. [36] studied 269 cryptographic vulnerabilities, finding that only 17% of the vulnerabilities they studied originated inside the cryptographic libraries, with the majority coming from improper uses of the libraries or interactions with other codebases. However, recent studies show that about 27% of vulnerabilities in cryptographic software are cryptographic issues, and the rest are system-level issues, including memory corruption and interactions with the host or other applications/libraries [17].

Hence, we modified OpenSSL to utilize MicroGuards for protecting private keys from potential information leakage by storing the keys in protected memory pages inside a single MicroGuard or multiple MicroGuards assigned per private key. Using multiple MicroGuards provides stronger security while adding more overhead due to the cost of caching MicroGuards.

To enable MicroGuards inside OpenSSL, all the data structures that store private keys such as `EVP_PKEY` needed protected heap memory allocation. This meant replacing `OpenSSL_malloc` wit `mg_malloc` and using `mg_mmap` at the initialization phase for creating one or multiple (per session) MicroGuards to store private keys. After storing the keys, access to MicroGuards is disabled by calling `mg_lock`. Only trusted functions that require access to private keys (e.g., `EVP_EncryptUpdate` or `pkey_rsa_encrypt/decrypt`) can access MicroGuards by calling `mg_unlock`. Modifying OpenSSL required fairly small code changes, and added 281 lines-of-code.

We measured the performance overhead of MicroGuards-enabled OpenSSL by evaluating it on the Apache HTTP server (httpd) that uses OpenSSL to implement HTTPS. Figure 5 shows the overhead of ApacheBench httpd with both the original OpenSSL library and the secured one with MicroGuards. ApacheBench is launched 100 times with various request parameters. We choose the TLS1.2 DHE-RSA-AES256-GCM-SHA384 algorithm with 2048-bit keys as a cipher suite in the evaluation.

The results show that on average MicroGuards introduces 0.47% performance overhead in terms of latency when using a single MicroGuard for protecting all keys, and 3.67% overhead when using a separate MicroGuard per session key. In the single MicroGuard case, the negligible overhead is mainly caused by in-kernel data structure maintenance for enforcing privilege separation and handling MicroGuards metadata. In the multiple-MicroGuards case, since httpd utilizes more than 16 MicroGuards (allocates a new MicroGuard per session), it causes higher overhead due to the caching costs within the kernel.

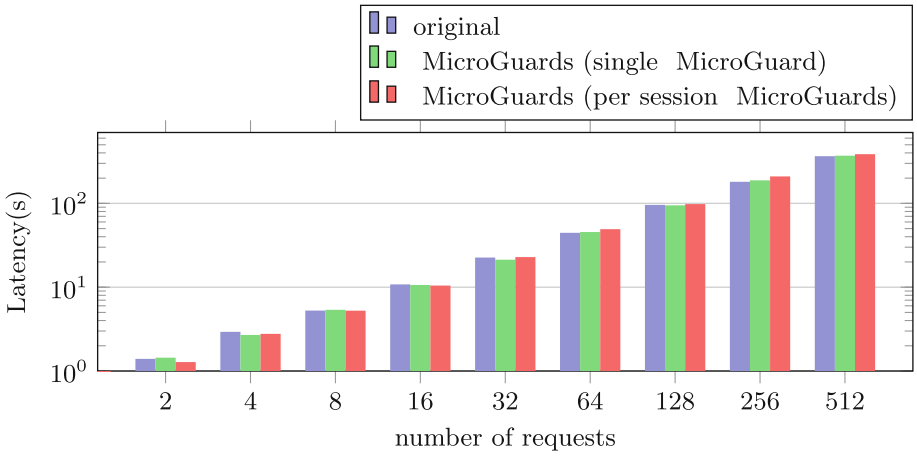

Fig. 5. Overhead of httpd on unmodified OpenSSL vs MicroGuards-enabled one.

5.3 LevelDB

Google's LevelDB is a fast key-value store and storage engine used by many applications as a backend database. It supports multithreading for both concurrent writers to safely insert data into the database as well as concurrent read to improve its performance. However, there is no privilege separation between threads, so each could have its private content isolated from other threads. We modified LevelDB to evaluate performance overhead of using the MicroGuards threading model when each thread has its own private storage that cannot be accessed by other threads.

We replaced the LevelDB threading backend (`env_posix`) that uses pthreads with MicroGuards-aware threading, where each thread creates an isolated MicroGuard as its private storage and computation. We used the LevelDB `db_bench` tool (without modification) for measuring the performance overhead of MicroGuards.

We generate a database with 400K records with 16-byte keys and 100-byte values (a raw size of 44.3MB). The number of reader threads is set to 1, 2, 4, 8, 16, and 32 threads for each successive run. The threads operate on randomly selected records in the database. The results in Figs. 6 and 7 show how multithreading can improve the performance of LevelDB, and utilising MicroGuards adds a small overhead on write (5%) and read (1.98%) throughput. As with OpenSSL previously, modifying LevelDB required only adding 157 lines-of-code around the codebase.

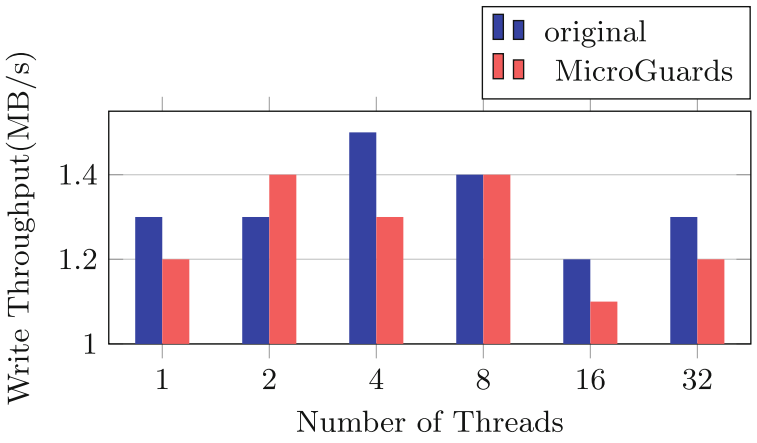

Fig. 6. LevelDB: performance overhead ofMicroGuards-based multithreading compare to pthread-based in terms of write throughput (5%).

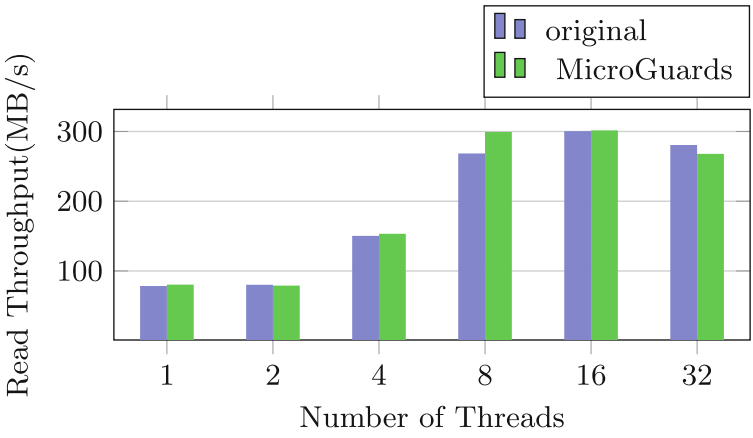

Fig. 7. LevelDB: performance overhead ofMicroGuards-based multithreading compare to pthread-based in terms of read throughput (1.98%).

6 Discussion and Conclusion

We have shown that MicroGuards provides a practical and efficient mechanism for intra-process isolation and inter-thread privilege separation on data objects. It adds small performance overhead and minimal memory footprint, which in essential for mobile and resource-constrained devices. However, the mechanism can still be taken further.

6.1 Address Space Protection Limitations

For single-threaded scenarios (e.g., event-driven servers), although MicroGuards can protect sensitive content from unsafe libraries or untrusted parts of the applications, it can be vulnerable if the untrusted modules are also MicroGuards-aware and already use the MicroGuards APIs. The application can use `mg_get` to query MicroGuard information and use the API to access them. This is not an issue when the untrusted code is running in a separate thread since the kernel does not provide it the capabilities required for accessing the other MicroGuards. It should be possible to modify popular event-driven libraries (e.g., libuv) to use threads purely to separate sensitive information such as key material, but we have not yet implemented this.

Various covert attacks [47] and side-channel attacks such as Meltdown [37] and Spectre [32] demonstrate how hardware and kernel isolation can be bypassed [30]. MicroGuards are currently vulnerable to these class of attacks, although the existing countermeasures within the Linux kernel are sufficient protection. We believe these types of attacks are important security threats, and hardening MicroGuards against them could be significant future work.

6.2 Compatibility Limitations

Providing a solution that is compatible with various operating systems and heterogeneous hardware is challenging. Though we picked our base kernel on Linux and built the abstraction with minimal dependencies, some application modification is still required. We believe that building more compatibility layers into our existing userspace implementation is possible and are open-sourcing our code to gather further feedback and patches from the relevant upstream projects we have modified.

Although Linux is the most widespread general-purpose kernel for embedded devices, as well as being the base for Android, still many even smaller devices depend on operating systems such as FreeRTOS. These often use ARM Cortex-M based hardware features for isolation (such as memory protection units (MPUs) [8,54]), or more modern CPUs with memory tagging extension [11]. We plan to explore the implementation of the MicroGuards kernel memory management on these single-address space operating systems, as well as broadening the port to Intel and PowerPC architectures on Linux (where the memory domains support is generally simpler to use than on ARM).

7 Related Work

There are many software or hardware-based techniques for providing process and in-process memory protection.

OS/Hypervisor-Based Solutions: Hardware virtualization features are used for in-process data encapsulation by Dune [14] by using the Intel VT-x virtualization extensions to isolate compartments within user processes. However, overall, the overheads of such virtualization-based encapsulation are more heavy-weight than MicroGuards. ERIM [55], light-weight contexts (lwCs) [38] and secure memory views (SMVs) [29] all provide in-process memory isolation and have reduced the overhead of sensitive data encapsulation on x86 platforms. The MicroGuards provides stronger security guarantees and privilege separation, allows more flexible ways of defining security policies for legacy code – e.g., without the use of threads as in our OpenSSL example, its small memory footprint makes it suitable for smaller devices, and it takes advantage of efficient virtual memory tagging by using hardware domains to reduce overhead. Burow et al. [19] leverage the Intel MPK and memory protection extensions (MPX) to efficiently isolate the shadow stack. Our efforts to provide an OS abstraction for in-process memory protection is orthogonal to these studies, which all have potential use cases for MicroGuards. Our focus has also been on lowering the resource cost to work well on embedded and IoT devices, while these projects are also currently x86-only. HiStar [59] is a DIFC-based OS that supports fine-grained in-process address space isolation, which influenced our work, but we focused on providing a more general-purpose solution for small devices by basing our work on the Linux kernel instead of a custom operating system. Flume [34] proposed process-level DIFC as a minimal extension to the Linux kernel, making DIFC work with

the languages, tools, and OS abstractions already familiar to programmers. It also introduced a cleaner label system (which HiStar have later adopted). Likewise, other DIFC-based systems only support per-process protection. They also add large overhead [34,57] or need specific programming language support [45]. MicroGuards, however, do no aim to enforce dataflow protection on all system objects, but only focuses on threads and address space objects to enable very lightweight privilege separation.

Compiler and Language Runtime: Various compiler techniques introduce memory isolation as part of a memory-safe programming language. These approaches are fine-grained and efficient if the checks can be done statically [24]. However, such isolation is language-specific, relies on the compiler and runtime, and not effective when applications are co-linked with libraries written in unsafe languages. MicroGuards abstractions are fine-grained enough to be useful to these tools, for example, to isolate unsafe bindings. Software fault isolation (SFI) [46,56] uses runtime memory access checks inserted by the compiler or by rewriting binaries to provide memory isolation in unsafe languages with substantial overhead. Bounds checks impose overhead on the execution of all components (even untrusted ones), and additional overhead is required to prevent control-flow hijacks, which could bypass the bounds checks [33]. ARMLock [62] is an SFI-based solution that offers lower overhead utilizing ARM MDs. Similarly, Shreds [20] provides new programming primitives for in-process private memory support. MicroGuards also uses ARM MDs for improving the performance of intra-process memory protection, but is a more flexible solution for intra-process privilege separation; it provides a new threading model for dynamic fine-grained access control over the address space with no dependency on a binary rewriter, specific compiler or programming language.

Hardware-Enforced Techniques: A wide range of systems use hardware enclaves/TEEs such as Intel's SGX [7] or ARM's TrustZone [10] to provide a trusted execution environment for applications that against malicious kernel or hypervisor [12,26,28,40,52]. The trust model exposed by these hardware features is very fixed, and usually results in porting monolithic codebases to execute within the enclaves. Hence, there are wide ranges of attack vectors, which many are memory vulnerabilities inside enclaves or their untrusted interface, in such systems [48,50]. EnclaveDom [39] utilizes Intel MPK to provide in-enclave privilege separation. MicroGuards provide better performance and more general solutions with no dependency on these hardware features; hence it can be used for in-enclave isolation and secure multi-threading to improves both security and performance of enclave-assisted applications [51]. Ultimately, dedicated hardware support for tagged memory and capabilities would be the ideal platform to run MicroGuards on [60]. We are planning on supporting more of these hardware features as future work, with a view to analyzing if the overall increase in hardware complexity offsets the resource usage in software for embedded systems.

References

1. Format string vulnerability in the Cherokee. https://www.cvedetails.com/cve/CVE-2004-1097/. Accessed 5 Jan 2020
2. IoT developer survey 2019. https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf
3. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b
4. Cyber security breaches survey 2018 (2018). https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2018
5. List of data breaches (2018). https://en.wikipedia.org/wiki/List_of_data_breaches
6. Almohri, H.M., Evans, D.: Fidelius charm: isolating unsafe rust code. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, pp. 248–255. ACM (2018)
7. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, vol. 13. ACM New York (2013)
8. ARM: CMSIS-Zone. https://arm-software.github.io/CMSIS_5/Zone/html/index.html
9. ARM: Architecture reference manual; ARMv7-A and ARMv7-R edition (2012). https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf. Accessed 26 May 2020
10. ARM: ARM®v8-M Security Extensions: requirements on development tools (2015)
11. ARM: ARM architecture reference manual ARMv8, for ARMv8-A architecture profile documentation (2018). https://developer.arm.com/docs/ddi0487/latest. Accessed 26 May 2020
12. Arnautov, S., et al.: SCONE: secure Linux containers with Intel SGX. In: OSDI, vol. 16, pp. 689–703 (2016)
13. Baumann, A., Appavoo, J., Krieger, O., Roscoe, T.: A fork () in the road. In: Proceedings of the Workshop on Hot Topics in Operating Systems, pp. 14–22. ACM (2019)
14. Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., Kozyrakis, C.: Dune: safe user-level access to privileged CPU features. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), pp. 335–348 (2012)
15. Berger, E.D., Zorn, B.G., McKinley, K.S.: Composing high-performance memory allocators (2001)
16. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: splitting applications into reduced-privilege compartments. In: USENIX Association (2008)
17. Blessing, J., Specter, M.A., Weitzner, D.J.: You really shouldn't roll your own crypto: an empirical study of vulnerabilities in cryptographic libraries. arXiv preprint arXiv:2107.04940 (2021)
18. Brumley, D., Song, D.: Privtrans: automatically partitioning programs for privilege separation. In: USENIX Security Symposium, pp. 57–72 (2004)
19. Burow, N., Zhang, X., Payer, M.: SoK: shining light on shadow stacks. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 985–999. IEEE (2019)
20. Chen, Y., Reymondjohnson, S., Sun, Z., Lu, L.: Shreds: fine-grained execution units with private memory. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 56–71. IEEE (2016)

21. Cox, G., Bhattacharjee, A.: Efficient address translation for architectures with multiple page sizes. ACM SIGOPS Operating Syst. Rev. **51**(2), 435–448 (2017)
22. Deng, Z., Saltaformaggio, B., Zhang, X., Xu, D.: iRiS: vetting private API abuse in iOS applications. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 44–56. ACM (2015)
23. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference, pp. 475–488. ACM (2014)
24. Elliott, A.S., Ruef, A., Hicks, M., Tarditi, D.: Checked C: making C safe by extension. In: 2018 IEEE Cybersecurity Development (SecDev), pp. 53–60. IEEE (2018)
25. Ferraiuolo, A., Zhao, M., Myers, A.C., Suh, G.E.: HyperFlow: a processor architecture for nonmalleable, timing-safe information flow security. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1583–1600. ACM (2018)
26. Frassetto, T., Gens, D., Liebchen, C., Sadeghi, A.R.: JITGuard: hardening just-in-time compilers with SGX. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2405–2419. ACM (2017)
27. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: KASLR is dead: long live KASLR. In: Bodden, E., Payer, M., Athanasopoulos, E. (eds.) ESSoS 2017. LNCS, vol. 10379, pp. 161–176. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62105-0_11
28. Guan, L., et al.: TrustShadow: secure execution of unmodified applications with ARM TrustZone. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pp. 488–501. ACM (2017)
29. Hsu, T.C.H., Hoffman, K., Eugster, P., Payer, M.: Enforcing least privilege memory views for multithreaded applications. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 393–405. ACM (2016)
30. Hunt, T., Jia, Z., Miller, V., Rossbach, C.J., Witchel, E.: Isolation and beyond: challenges for system security. In: Proceedings of the Workshop on Hot Topics in Operating Systems, pp. 96–104. ACM (2019)
31. Intel: Intel® 64 and IA-32 architectures software developer's manual (2019). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
32. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. arXiv preprint arXiv:1801.01203 (2018)
33. Koning, K., Chen, X., Bos, H., Giuffrida, C., Athanasopoulos, E.: No need to hide: protecting safe regions on commodity hardware. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 437–452. ACM (2017)
34. Krohn, M., et al.: Information flow control for standard OS abstractions. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 321–334. ACM (2007)
35. Lamowski, B., Weinhold, C., Lackorzynski, A., Härtig, H.: Sandcrust: automatic sandboxing of unsafe components in Rust. In: Proceedings of the 9th Workshop on Programming Languages and Operating Systems, pp. 51–57. ACM (2017)
36. Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why does cryptographic software fail? A case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems, pp. 1–7 (2014)
37. Lipp, M., et al.: Meltdown. arXiv preprint arXiv:1801.01207 (2018)
38. Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., Druschel, P.: Light-weight contexts: an OS abstraction for safety and performance. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 49–64 (2016)

39. Melara, M.S., Freedman, M.J., Bowman, M.: EnclaveDom: privilege separation for large-TCB applications in trusted execution environments. arXiv preprint arXiv:1907.13245 (2019)
40. Mo, F., Tarkhani, Z., Haddadi, H.: SoK: machine learning with confidential computing. arXiv preprint arXiv:2208.10134 (2022)
41. Morgan, L.: List of data breaches and cyber attacks in October 2017 – 55 million records leaked (2017). https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-october-2017-55-million-records-leaked/
42. Morris, J., Smalley, S., Kroah-Hartman, G.: Linux security modules: general security support for the Linux kernel. In: USENIX Security Symposium, Berkeley, CA, pp. 17–31. ACM (2002)
43. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: Libmpk: software abstraction for Intel memory protection keys. arXiv preprint arXiv:1811.07276 (2018)
44. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: USENIX Security Symposium (2003)
45. Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: practical fine-grained decentralized information flow control, vol. 44. ACM (2009)
46. Sehr, D., et al.: Adapting software fault isolation to contemporary CPU architectures (2010)
47. Sigurbjarnarson, H., Nelson, L., Castro-Karney, B., Bornholt, J., Torlak, E., Wang, X.: Nickel: a framework for design and verification of information flow control systems. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 287–305 (2018)
48. Singh, J., Cobbe, J., Quoc, D.L., Tarkhani, Z.: Enclaves in the clouds: legal considerations and broader implications. Commun. ACM **64**(5), 42–51 (2021)
49. StewardJack, J.: The ultimate list of internet of things statistics for 2022 (2021). https://findstack.com/internet-of-things-statistics/
50. Tarkhani, Z.: Secure programming with dispersed compartments. Ph.D. thesis, University of Cambridge (2022)
51. Tarkhani, Z., Madhavapeddy, A.: Enclave-aware compartmentalization and secure sharing with Sirius. arXiv preprint arXiv:2009.01869 (2020)
52. Tarkhani, Z., Madhavapeddy, A., Mortier, R.: Snape: the dark art of handling heterogeneous enclaves. In: Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking, pp. 48–53 (2019)
53. Tarkhani, Z., Qendro, L., Brown, M.O., Hill, O., Mascolo, C., Madhavapeddy, A.: Enhancing the security & privacy of wearable brain-computer interfaces. arXiv preprint arXiv:2201.07711 (2022)
54. Tock: Finer grained memory protection on Cortex-M3 MPUs. https://github.com/tock/tock/issues/1532
55. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: ERIM: secure, efficient in-process isolation with protection keys (MPK). In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1221–1238 (2019)
56. Wahbe, R., Lucco, S., Anderson, T.E., Graham, S.L.: Efficient software-based fault isolation. In: ACM SIGOPS Operating Systems Review, vol. 27, pp. 203–216. ACM (1994)
57. Wang, J., Xiong, X., Liu, P.: Between mutual trust and mutual distrust: practical fine-grained privilege separation in multithreaded applications. In: 2015 USENIX Annual Technical Conference (USENIX ATC 2015), pp. 361–373 (2015)

58. Watson, R.N., et al.: Cheri: a hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 20–37. IEEE (2015)
59. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 263–278. USENIX Association (2006)
60. Zeldovich, N., Kannan, H., Dalton, M., Kozyrakis, C.: Hardware enforcement of application security policies using tagged memory. In: OSDI, vol. 8, pp. 225–240 (2008)
61. Zero, P.: Introduction: Bugs in memory management code (2019). https://googleprojectzero.blogspot.com/2019/01/taking-page-from-kernels-book-tlb-issue.html
62. Zhou, Y., Wang, X., Chen, Y., Wang, Z.: ARMlock: hardware-based fault isolation for ARM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 558–569. ACM (2014)