

USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

DisARMed: Attacking ARM TrustZone from Userspace with Memory Aliasing

Jacqueline Henes, Matthew Bowden, and Mihai Ordean, *University of Birmingham;*
David Oswald, *Durham University*

https://www.usenix.org/conference/woot26/presentation/henes

This paper is included in the Proceedings of the
20th USENIX WOOT Conference on Offensive Technologies.

August 10–11, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-57-1

Open access to the Proceedings of the
20th USENIX WOOT Conference on Offensive Technologies
is sponsored by USENIX.



DisARMed: Attacking ARM TrustZone from Userspace with Memory Aliasing

Jacqueline Henes[*]
jkh703@student.bham.ac.uk
University of Birmingham

Matthew Bowden[*]
mxb1143@student.bham.ac.uk
University of Birmingham

Mihai Ordean
m.ordean@bham.ac.uk
University of Birmingham

David Oswald
david.f.oswald@durham.ac.uk
Durham University

Abstract

Modern systems security relies on memory isolation mechanisms like trusted execution environments and kernel privilege separation to enforce strong isolation boundaries. However, many of these mechanisms place implicit trust in system memory, leaving them open to hardware attacks on external DRAM. In this paper, we introduce DISARMED, an attack on ARM processors that exploits memory aliasing techniques from userspace, compromising both the Linux kernel and ARM TrustZone. We demonstrate for the first time that memory aliasing attacks are practical for adversaries that do not have access to the kernel. We additionally implement and evaluate a mitigation for DISARMED, using a lightweight alias detection mechanism. Our solution has minimal impact on boot time of about one second. Finally, we discuss the wider applicability of DISARMED, considering other relevant potential attack vectors, applicable memory massaging techniques, and security mechanisms that may be affected.

1 Introduction

Virtual memory and architectural privilege levels provide strong isolation properties that kernels, such as Linux, rely on to isolate user programs. When additional isolation is required, trusted execution can be used. ARM TrustZone [1] is a widely deployed technology that provides architectural separation for Trusted Execution Environments (TEEs); it is designed such that malicious operating systems are not able to compromise the confidentiality and integrity of the data in TrustZone. It underlies the Android KeyStore, which stores cryptographic keys for user software and, for platforms such as Qualcomm's TEE, is used as a hardware-backed secure storage for applications [14].

In this paper, we study the security of both TrustZone and the Linux kernel in the light of recent attacks that exploit memory aliasing. Specifically, De Meulemeester et al. undermined AMD SEV-SNP by using malicious DRAM, dubbed "BadRAM", to break remote attestation and hence VM security guarantees [15]. TrustZone has similar goals to SEV-SNP in maintaining the confidentiality of data in secure memory while trusting DRAM. This makes it an attractive target for a physical memory aliasing attack.

De Meulemeester et al. focused on TEEs, starting with an attacker that already has full control over the Rich Execution Environment (REE); this assumes that an attacker has already defeated mechanisms such as a secure boot chain and kernel memory protection beforehand. In this paper, we introduce a novel *end-to-end* attack *i.e.*, an attack that escalates from userspace, to the Linux kernel, and into the TEE. This attack, which we call DISARMED, uses memory aliasing to defeat not just TrustZone but also the virtual memory protections designed to isolate userspace processes.

DISARMED explores the effects of memory aliasing on ARM platforms and the implications for solutions relying on TrustZone. It is, to our knowledge, the first effective aliasing attack against the Linux kernel originating in userspace, breaking fundamental assertions about memory access and integrity. Our proposed attack enables adversaries to attack TrustZone and the Linux kernel even in the presence of a typical secure boot chain, as there is no need to patch signed boot data—such as the kernel, device tree, and bootloader [17]—on disk. Instead, they can be patched in-memory from userspace, bypassing secure-boot protections. Prior memory aliasing attacks are limited to platforms without a secure boot chain; by eschewing this requirement DISARMED represents a practical step in weakening the required attacker model for memory aliasing attacks.

We additionally introduce the concept of *alias-tolerant* memory-massaging to describe the requirements needed to overcome the unique constraints involved in such an attack from userspace. These aliasing-induced constraints frustrate many existing memory-massaging techniques, which often depend on strategies that cause system instability under aliasing. We consider the memory content and layout from the perspectives of DRAM, physical address space, and the virtual address space of the attacking process. We use this to inform

[*]These authors contributed equally to this work.

the attack strategy and expand upon the current understanding of the memory aliasing primitive. By implementing the first *alias-tolerant* memory-massaging strategy, we demonstrate how to bias Linux page allocation in a reliable manner, making DISARMED possible.

1.1 Contributions

In summary, the main contributions of this paper are:

- We propose DISARMED, the first *end-to-end* attack using memory aliasing that gives an unprivileged userspace adversary full read-write access to all regions of DRAM, including kernel and TrustZone memory regions.
- We demonstrate this access by showing that DISARMED can be used to escalate privilege when attacking the kernel and retrieve TEE shielded PKCS#11 secrets when attacking TrustZone.
- We show memory aliasing on ARM platforms for the first time and characterize how DIMM Serial Presence Detect (SPD) modification behaves across multiple ARMv8-A platforms.
- We develop an *alias-tolerant* memory massaging technique that allows an adversary to allocate pages physically aliased to target memory without accidentally overwriting critical system data, which can render a system inoperable; this is otherwise not possible with traditional memory massaging techniques.
- Finally, we propose and demonstrate an efficient mitigation strategy that adds a minimal delay to the boot process ($\sim$ 1 s). We implement and benchmark this mitigation in Trusted Firmware-A (TF-A) to detect the presence of aliasing at boot time.

Responsible Disclosure We disclosed our results to ARM in February 2025. The details of the disclosure, including ARM's response, can be found in Appendix A.

Open Science To allow our work to be reproduced as well as encourage future research on DRAM aliasing attacks, we include a number of open-sourced artifacts as part of our work. The collection includes our proof-of-concept attack code, prototype mitigation, and related software tooling. See https://github.com/bhamsec/disarmed and Appendix B for more details.

1.2 Outline

The remainder of this paper is structured as follows: First we discuss related work regarding physical memory aliasing, memory massaging, and attacks against TrustZone in Section 2. Then, in Section 3 we introduce DISARMED, our userspace memory aliasing attack against the Linux kernel and TrustZone. We discuss the adversary model, relevant required background, and the challenges that dictated the constraints when designing DISARMED. We go on to describe the implementation of our attack in Section 4 and evaluate the efficacy of its application against multiple ARMv8-A platforms in Section 5. Finally, we describe mitigations in Section 6, implementing one in firmware as a proof-of-concept design. In Section 7, we elaborate on the potential for future exploits using aliasing as a primitive, consider relevant secure compartmentalization technologies, and potential alternative methods of inducing aliasing on different platforms.

2 Related Work

Physical Memory Aliasing De Meulemeester et al. introduced physical memory aliasing via modified SPD Electrically Erasable Programmable Read-Only Memories (EEPROMs) [15] and showed how DRAM addressing configurations stored on DIMM SPD chips can be modified to report larger, incorrect memory capacities, resulting in an aliasing effect. Using this, the authors showed that memory aliasing can be used to break AMD SEV-SNP attestation. De Meulemeester et al. additionally demonstrated the use of a low-cost physical DIMM interposer to induce a different memory aliasing effect [16] that can be toggled on and off during runtime. This allows for some existing memory aliasing mitigations to be bypassed, enabling an attack on Intel SGX. Louka et al. additionally demonstrate the effect of physical memory aliasing on Keystone, a TEE built on top of RISC-V PMP [31].

DIMM Serial Presence Detect SPD is the mechanism for determining the properties of a DIMM, such as its size and configuration. On each DIMM is a serial EEPROM which stores this information; this is parsed by firmware so that the DIMMs can be used. This includes reading the amount of memory available (*density*), but also memory timings, the number of banks and rows, a serial number, the manufacture date, and related DIMM metadata. This mechanism is standardised by JEDEC [24].

BadRAM introduced modification of this SPD data as a strategy for inducing aliasing, a condition in which multiple physical addresses map to a single on-DIMM address [15]. We take the same approach for DISARMED, and find that it is also effective on ARMv8-A platforms that support discrete DIMMs. Our approach similarly requires one-time physical access to the target DIMM, however, in contrast to BadRAM, does not require root privileges. As a result, DISARMED can be executed against a target where secure boot is enforced.

DRAM Attacks Attacks using DRAM as the attack vector have been well explored—so-called practical 'cold-boot'

attacks stealing encryption keys were demonstrated by Halderman et al. [22], and more recent work shows electromagnetic fault injection on SO-DIMM pins for local privilege-escalation [13]. Next, we focus specifically on the "Rowhammer" class of attacks and the physical memory massaging techniques they employ.

Rowhammer-based Attacks Modern DRAM is prone to reliability errors induced by electrical charge disturbance. Kim et al. show that repeated accesses to one row could affect the charge of cells in neighboring rows, effectively 'flipping' bits without ever performing a read/write operation to the row in question [27]. This gave rise to a class of attacks exploiting this bug, eventually known as Rowhammer. In [41], the Project Zero team were among the first to demonstrate a privilege escalation exploit from a unprivileged userspace context. In the following years, variations upon the original Rowhammer primitive were published, exploring a breadth of possible attack targets [28, 40, 51] and also innovations on the method of inducing bit-flips [20, 48, 58]. In response to these attacks, information about the physical memory layout, such as the page mappings for the current process, has been restricted for unprivileged users [44].

Finally, Rowhammer-style DRAM attacks have rarely targeted TEEs. However, Jang et al. used Rowhammer to flip bits in encrypted enclave memory to perform a Denial-of-Service (DoS) attack against Intel SGX [23].

Physical Memory Massaging Previous work on Rowhammer uses a variety of techniques to manipulate memory in order to execute specific attacks. Broadly referred to as *memory massaging*, the aim of these techniques is to influence memory allocation such that allocations use predictable memory locations. An attacker can then use this predictability in the subsequent exploitation stage.

One common memory massaging technique utilized is *page-spraying*, introduced by Kemerlis et al. as part of the `ret2usr` exploit chain [26]. Page-spraying is a probabilistic approach that involving filling ('spraying') memory with specific data structures via memory allocation requests, with the goal of reducing non-determinism about the contents of memory. This technique is used to increase the probability that an arbitrary address points to an attacker-crafted payload [26], or to find address sets vulnerable to Rowhammer bit-flips via Page Table Entry (PTE) spraying [41].

Another technique used specifically for Rowhammer exploitation is *Phys Feng Shui*, detailed in [51]. Phys Feng Shui illustrates a more deterministic method of physical memory massaging that relies on influencing memory allocation by exhausting available memory. This can cause memory allocators to serve memory in predictable patterns due to system memory pressure. *Frame Feng Shui* [28] takes a similar approach, exploiting the page frame cache of the page allocator to target specific victim process pages. Rubicon [12] introduces primitives leveraging Linux page allocator per-CPU and free lists for a fully deterministic Rowhammer attack without relying on specific kernel features. Proposed mitigations against memory massaging include using guard pages to protect targeted data structures [52] or more robust memory allocator designs to prevent the massaging altogether [12, 43].

ARM TrustZone Attacks Many existing attacks against ARM TrustZone leverage classical exploitation techniques against TEE implementations, such as buffer overflows [10], out-of-bounds writes [9], and unbounded `memcpy` [21], exacerbated by poor deployment of typical defences such as Address-space Layout Randomization (ASLR) and rollback prevention [11]. These attacks are dependent on bugs in TEE and trustlet implementations rather than architectural bypasses of TrustZone. In contrast, DISARMED attacks the architectural mechanisms enforcing TrustZone, making it agnostic to the TEE implementation on platforms where DRAM is not soldered. Physical fault injection exploits against TrustZone include VoltJockey [39] and CLKscrew [42], which both attack TrustZone on ARMv7-A by manipulating Dynamic Voltage and Frequency Scaling (DVFS) to induce faults in TrustZone code and break cryptographic algorithms (e.g., RSA and AES). DISARMED demonstrates broader TrustZone compromise, enabling full memory access rather than specific cryptographic attacks.

Gross et al. introduced Direct Memory Access (DMA) attacks against TrustZone, assuming a malicious hardware in an embedded FPGA [19]. DISARMED similarly makes use of a malicious peripheral to compromise TrustZone, although it allows for a less sophisticated attacker and is not limited to System-on-Chips (SoCs) with FPGA peripherals.

3 The DISARMED Attack

Now, we introduce DISARMED, a userspace memory aliasing attack that targets both the Linux kernel and TrustZone. DISARMED works by biasing the Linux page allocator to enable unprivileged userspace software to map, read, and modify physical memory. It requires two primitives:

1. Physical Memory Aliasing
2. *Alias-tolerant* Memory Massaging

First, an adversary uses SPD modification to induce physical memory aliasing by modifying the DIMM description stored on a flash chip, thus creating a DIMM that appears to have a larger capacity than its true capacity. Then, they use an *alias-tolerant* memory massaging strategy to obtain safe aliased memory regions. This is required to avoid overwriting kernel data via memory aliases, which causes the running system to fault and crash. With these two requirements satisfied, they are able to modify memory regions through aliasing whilst

bypassing kernel and MMU protections. Aliases to page tables that exist in attacker memory allocations can be used to remap physical pages into attacker-controlled address space.

In order to escalate privilege within the REE by attacking the kernel, an adversary maps and modifies kernel memory directly. By mapping aliases to memory that would normally be protected by TrustZone, they are additionally able to escalate privilege by reaching from the REE context into the TEE context. The remaining sections set out our adversary model, discuss relevant background, define the attack primitives in more depth, and describe what outcomes are possible as a result of DISARMED.

Adversary Model We assume an adversary with one-time physical access to a targeted system (to change SPD of a DIMM) and the ability to run code in unprivileged userspace context. This is a weaker adversary than in previous memory aliasing attacks [15], where one-time physical access and kernel access is assumed. Similarly, other TEE attacks often require root privileges [35, 42]. In particular, this enables attacks against targets with a secure boot chain where this was previously not possible. Examples of scenarios where physical access and userspace code execution is realistic include cloud service providers and embedded platforms.

Experimental Setup We tested and evaluated our attack on three Armv8-A platforms (see Table 1) that are representative of these use cases: the Ampere Altra DP corresponds to a typical server architecture, while the NXP LS1046ARDB is representative of an embedded device (e.g., in an industrial application). Finally, the ARM Morello SDP demonstrates the issue on an experimental platform with Capability Hardware Enhanced RISC Instructions (CHERI) capability support [7].

We surveyed a non-exhaustive selection of major cloud service providers to check what platforms are offered for Arm-based virtual machines and found that a significant proportion use Ampere Altra-based platforms for their host hardware. In the cases where something other than Ampere hardware was used, service providers typically use in-house Arm processor designs, such as AWS's Gravitron family of CPUs. Our choice of platforms thus broadly reflects the current hardware used for Arm workloads in cloud environments. A summary of this survey is shown in Table 2.

3.1 Preliminaries

To introduce DISARMED, we first provide an overview of the ARMv8-A architecture and the Linux kernel. The ARMv8-A architecture has two relevant features: TrustZone, which DISARMED seeks to bypass; and a virtual memory architecture, which it targets. Regarding Linux, DISARMED seeks to manipulate memory allocation procedures in order to safely and effectively make use of memory aliasing. Finally, we describe a common kernel exploitation strategy that can be combined with DISARMED.

Virtual Memory ARMv8-A uses a virtual memory scheme where pages, i.e., *granules*, of configurable size (4KB, 16KB, 64KB) are translated from a virtual address space to physical address space. This translation is done by walking a set of page tables, using each part of the virtual address as an offset into either the next table or output region [6]. In the typical 4KB granule configuration, each page table has 512 64-bit entries. Each of those entries is either (1) invalid or (2) a descriptor. Table descriptors point to the table used for the next level of translation, which will be indexed to obtain another descriptor. Page descriptors (only valid in level 3 translation tables) point to the single physical page that a virtual page maps to [6]. Finally, these look-ups are cached in the Translation Look-aside Buffer (TLB), a cache used to improve lookup performance. There are no guarantees that the contents of the TLB and main memory are coherent following modifications of page table entries, so it is necessary to flush the TLB after any modification. However, TLB flush instructions are not defined in the userspace context, thus preventing explicit TLB flushing from userspace [6].

ARM TrustZone ARM TrustZone is the technology backing trusted execution in ARMv8-A architectures [1]. It provides the architectural primitives required for a TEE to be isolated from the REE. The ARMv8-A exception model delineates execution states into *secure* and *non-secure* (or *normal*) *worlds*. Execution is split into these two worlds; this permits a REE to run in the non-secure world while a TEE runs in the secure world. Importantly, the secure and non-secure worlds share the same processor resources and memory. In order to prevent non-secure code surreptitiously gaining access to the secure world, the main address space is split into two: the NS (non-secure) bit—representing if the processor is currently in the secure or non-secure world—is sent to the MMU during a memory access [4]. The MMU uses this to selectively deny the non-secure world access to secure-only memory regions.

While the ARM TF-A threat model explicitly excludes the need to trust DRAM [29], in practice this is not feasible. Two of the three platforms we tested did not use these mechanisms. Indeed, ARM has designed and licensed AMBA peripherals, such as the TZC-380 [3] TrustZone Access Space Controller (TZASC) that are designed to use DRAM to back secure memory. Additionally, while TEEs such as OP-TEE support running from SRAM, it is a commonly supported configuration to run from TrustZone-protected DRAM [50]; the reason for this being that SRAM is simply too small. This is also considered acceptable by the GlobalPlatform TEE Protection Profile, which makes clear that partially-trusted off-SoC memory is acceptable [18].

System	Microarchitecture	Unaliased DIMMs	Type	Aliased Capacity
ARM Morello SDP	Morello (Neoverse-N1)	2×8GB MTA9ASF1G72PZ-3GE2TI	RDIMM	32GB
Ampere Altra DP	Neoverse-N1	2×8GB MTA9ASF1G72PZ-3GE2TI	RDIMM	32GB
NXP LS1046ARDB	Cortex-A72	1×16GB MTA18ADF2G72AZ-3GR1ZI	UDIMM	32GB

Table 1: Evaluation platforms used throughout this paper.

CSP Instance	Processor
All AWS EC2 Arm instances	Amazon Gravitron
Google Cloud A4X	NVIDIA Grace
Google Cloud C4A	Google Axion
Google Cloud N4A	Google Axion
Google Cloud Tau T4A	Ampere Altra
Oracle Cloud Ampere A1	Ampere Altra
Oracle Cloud Ampere A2	AmpereOne
Microsoft Azure Bpsv2	Ampere Altra
Microsoft Azure Dpsv5	Ampere Altra
Microsoft Azure Dpsv6	Azure Cobalt 100
Microsoft Azure Epsv5	Ampere Altra
Microsoft Azure Epsv6	Azure Cobalt 100

Table 2: Survey of Arm-based virtual machine instances [2, 32, 33, 34, 37]

The Linux Page Allocator The Linux kernel allocates paged memory to many different processes and systems—this is all managed by the Linux *page allocator*. The page allocator, also referred to as the *zone buddy allocator*, determines which physical pages are allocated when requests for memory are made [47].

Zones and Physical Memory Linux uses the abstraction of *zones* to delineate blocks of physical memory in an architecture independent manner [47]. The page allocator uses zones to support compatibility across different architectures, which set constraints on physical memory must be accessed. For example, the `ZONE_DMA` or `ZONE_DMA32` zones are typically located at the 'bottom' of physical memory. These zones represent ranges used by peripheral device DMA, particularly legacy devices not supporting 64-bit addressing. `ZONE_NORMAL` is normal memory that can be accessed at any time, *i.e.*, is where most allocations reside. Whether a zone is present on a certain system is determined by kernel configuration options, in line with architecture-specific requirements.

In Table 3, we show a list of all memory zone types and common characteristics, as well as which zones were enabled on the ARMv8-A configurations we tested. `ZONE_NORMAL` typically comes after `ZONE_DMA`/`ZONE_DMA32` (if enabled) and usually is the largest zone in a system. When allocating memory, given no specific constraints, the page allocator will prefer to allocate from `ZONE_NORMAL`, but can optionally allocate from other zones if a given zone reaches capacity.

Huge Pages In some use cases, the standard page size can limit the performance of applications that work with large amounts of data. The typical 4KB page can introduce noticeable overhead when performing large memory-intensive operations. These processes benefit from larger page sizes, and so Linux provides `hugepages` for processes with large memory overhead. Using larger page sizes broadly results in less TLB pressure: any TLB misses run faster, and there are less TLB misses overall, as more virtual memory is mapped per-page. On ARMv8-A, a 4KB granule configuration has two sizes of `hugepages` available: 2MB and 1GB.

Linux provides two methods to use huge pages: HugeTLB pages and Transparent Hugepages (THPs). HugeTLB pages provide a static means to ensure a pool of huge pages available for use [46]. Linux will reserve regions of memory used as huge pages. HugeTLB pages can be configured at boot time through kernel command-line parameters and at runtime via the `sysfs` interface. Userspace processes can then create HugeTLB mappings via `mmap` using `MAP_HUGETLB`. Alternatively, with THP, the kernel dynamically promotes runs of normal-sized pages to huge pages without reserving a pool of memory, providing a more flexible means for using huge pages [45].

Linux Kernel Exploitation For an attacker with the ability to freely modify kernel memory, attacks against the kernel are well understood; they range from patching kernel code [57] to modifying the `struct cred` that describes the current process [30, 51]. These allow for the breaking down of all inter-process and kernel security barriers.

One such attack is `modprobe_path` patching: when the kernel loads an executable, it may find that it does not have valid ELF magic. If the first four bytes of the executable contain one (or more) non-printable characters and all registered `binfmt` handlers are exhausted without match, the kernel will resort to using `modprobe` to load an appropriate module. To do this, it executes the `modprobe` executable in userspace as root in an attempt to search for an appropriate binfmt module [38]. This requires that the kernel knows the location of the `modprobe` executable in the filesystem.

By mapping kernel memory, an attacker can search for the existing `modprobe_path` and then patch it in memory to instead be an attacker controlled binary, e.g., `/tmp/exploit`.

Type	Typical Memory Range	Enabled?	Usage
ZONE_DMA	$<$ 16MB	●	Backward compatibility for legacy DMA devices.
ZONE_DMA32	$<$ 4GB	●	DMA support for devices that can only operate in 32-bit address spaces. 64-bit only.
ZONE_NORMAL	16MB – 896MB(32-bit) $>$ 4GB(64-bit)	●	Main memory pool allocating normal, regular pages. DMA operations can use this zone if the device does not have address space constraints.
ZONE_HIGHMEM	$>$ 896MB	✖	Utility zone for physical memory not covered by permanent kernel mappings. 32-bit only.
ZONE_MOVABLE	Variable	✖	Like ZONE_NORMAL, but page contents are movable *i.e.*, content may move between different physical pages.
ZONE_DEVICE	Device-specific	◐	Representative of specific device memory e.g., GPU RAM.

Table 3: Linux memory zones and the typical regions of physical memory they encompass. ● indicates a zone is enabled for all platforms tested, ✖ indicates it was disabled, and ◐ shows that it was enabled on one platform but disabled on the others. Different kernel configurations have different zones enabled depending on required use cases, e.g., the Ampere Altra DP had ZONE_DEVICE enabled, as opposed to the rest of the tested platforms.

There, they place an executable that copies the system shell, change its ownership to root, and set the setuid bit. Finally, they can execute an arbitrary binary that begins with 0xFFFFFFFF to trigger the kernel to execute /tmp/exploit as root. This places a setuid root shell on the filesystem, which can be run by any user to get a root shell.

3.2 Userspace Primitives

Here we define the exploit primitives required to implement an attack that uses memory aliasing from userspace, and the particular challenges arising as a result.

Physical Memory Aliasing DISARMED crucially relies on malicious physical memory aliasing occurring on a target system, where read/write operations to one memory address are mirrored on another memory address. The specific aliasing behavior required for this attack is where for each DRAM address there are exactly two corresponding physical addresses, *doubling* the amount of perceived installed memory.

***Alias-Tolerant* Memory Massaging** Existing kernel mitigations limit the knowledge a userspace attacker has about physical memory. Typical memory-based attacks consequently use memory massaging primitives to influence memory allocation to their benefit. The memory massaging primitive not only needs to be usable from userspace, but must also work in tandem with the aliasing primitive. This latter requirement introduces design considerations specific to aliased systems, requiring novel *alias-tolerant* memory massaging techniques.

3.3 Design Challenges

DISARMED stands out from previous work because it is executed entirely from unprivileged userspace. The following section presents the additional challenges that must be considered compared to prior attacks.

Memory Addressing Prior memory aliasing attacks have visibility of the physical address space and are able to map memory as needed because they execute in the kernel [15]. In contrast, DISARMED is subject to the behavior of the kernel page allocator. This presents three main issues:

1. *Allocation Control*: Code running in userspace cannot request specific physical pages of memory; instead, the kernel page allocator decides which physical pages are mapped. This prevents straightforward aliasing attacks, where an alias to some target address is calculated and then mapped.

2. *Allocation Knowledge*: When the kernel allocates memory, a userspace process has no knowledge where virtual pages reside in physical memory. This, in turn, makes it impossible to do many allocations and then check if any are mapped to a desired physical memory range.

3. *Allocation Usage*: When a page of memory is allocated to a process, a corresponding physical page is not made resident in memory immediately. Instead, the virtual page is mapped to the zero page. When a write to this page occurs, the kernel performs Copy-on-Write (CoW): a physical page is allocated to back the virtual page, zeroed, and then populated with the result of the data write. This means individual virtual pages become resident in memory only after an explicit write, not when they are allocated. Naturally, the page will overwrite any existing memory at the location it is written to. This means that it is not possible to only allocate large swathes of memory and scan it; it will always read zero. Additionally, it also means that it is not possible to selectively *skip* memory

regions. The next write to any non-resident virtual page, regardless of where that page is in the virtual address space, will always be mapped to the same physical page.

System Stability Prior attacks mask off aliased regions of memory from general-purpose use for system stability, which requires root access [15]. Without this capability, DISARMED instead risks that every allocation maps memory aliased to kernel data. As described, it is not possible to read this memory, nor is it possible to choose *not* to overwrite this memory. Even if the page allocator can be predicted, unfavourable allocation offers no recourse.

Memory Layout Additionally, the layout of memory is hazardous itself: aliases of kernel data sit in between many aliased pairs of addresses (see Figure 4). When this is combined with typical Linux page allocation behaviour, it becomes very likely that with a naive allocation strategy kernel data will be overwritten *before* useful aliasing is discovered. Where previously control over allocations would have been a luxury, some level of control is now a necessity.

Memory Massaging These constraints leave the attacker with quite a weak primitive: they can write values, but they cannot control or know where; they can only write entire pages; and they cannot read anything before it is written. These pages that they write can potentially overwrite critical kernel data, crashing the system. Typically this is where memory massaging techniques are used by an attacker; however many existing techniques use strategies that are not suitable in an environment with memory aliasing. For example, memory exhaustion, a common strategy mapping almost all available memory to force predictable allocation behaviour, will certainly cause system crashes via kernel data overwrites.

Solutions To solve these issues, we describe and implement the first *alias-tolerant* memory massaging strategy:

1. *Allocation Control*: By making use of HugeTLB allocations, an attacker is able to exert limited but sufficient spatial control over page allocation. This makes it possible to work around the hazardous memory layout, mitigating stability concerns.

2. *Allocation Knowledge*: The attacker aims to control *any* of their own page tables, as opposed to a specific target. Through page table spraying, a suitable victim can be located in large swathes of memory, avoiding the need for precise knowledge of where allocations are.

3. *Allocation Usage*: Rather than attempting to use aliasing to overwrite victim memory, DISARMED instead causes the allocation of many page tables and waits for them to overwrite its own memory. As a result, resident pages in the attacker process now alias to victim data, and can be modified without CoW or zero-page issues.

By using alias-tolerant techniques, DISARMED can construct a physical memory read/write capability, previously only achieved with root access, avoiding system instability.

3.4 DISARMED Outcomes

With DISARMED, an attacker can read from and write to physical memory addresses without limitation. Attackers are also able to bypass typical range-delimited memory protection such as TrustZone's secure memory. This results in an end-to-end system compromise from unprivileged userspace. In the following we provide a high-level description of the capabilities an attacker gains over the kernel and TrustZone when using DISARMED.

Kernel Compromise The ability to read/write physical memory lends itself naturally to attacks against the running kernel. Attackers can scan memory to find where kernel data is located, which can then be patched depending on the goal of the attacker; often with escalation to root privileges being an end goal; cf. Section 3.1 for examples.

TrustZone Compromise Attacking TrustZone requires an additional exploitation step when using the memory aliasing primitive. This is because the read/write capability, by itself, is not sufficient, as direct accesses to secure memory would be denied from the normal world. However, accesses via aliases to secure memory are not prevented. As such an attacker can access TrustZone secure memory by mapping the corresponding aliases into their address space. When a memory access to an alias of secure memory occurs, the architecture checks if the memory access from the non-secure world is within a secure memory region and hands out the contents. The secure memory region locations are typically fixed for a given platform (and are often public knowledge), so finding the region's alias is straightforward when physical addressing is possible. This approach allows for a full compromise of both the integrity and the confidentiality of TrustZone.

4 Defeating Memory Protection with DISARMED

Next, we describe the implementation details of DISARMED to attack both the Linux kernel and TrustZone on ARMv8-A. We start by inducing/characterizing aliasing behavior and then go on to describe our implementation of *alias-safe* memory massaging. Finally, we describe how this massaging can be used to construct a physical memory read-write capability.

4.1 Memory Aliasing on ARMv8

SPD Aliasing The first primitive required by DISARMED is a memory aliasing method, realized by modifying the SPD data on a DIMM. We connect the I^2C interface of a Raspberry Pi Pico to the exposed SMBus interface on a DDR4 DIMM socket for SPD data to be read/written over I^2C. We modify the SPD data of our DIMM, doubling the density (*i.e.*, 8GB to 16GB, or 16GB to 32GB) and increasing number of row addresses by one. While some DIMMs do not enable write protection [15], many do as it is required for JEDEC compliance [24]. To disable the write protection, we use the Raspberry Pi Pico to send the Clear All Write Protection (CWP) instruction while the SA0 pin is connected to a 9 V supply. We then write the modified SPD data back to the DIMM over I^2C to complete our modification, and install the modified DIMM(s) into the target platforms. As a result, the firmware and OS of the target believe that the memory installed is twice the true DIMM capacity.

Alias Discovery To understand ARMv8-A's aliasing semantics in a environment with reduced complexity, we implemented an alias-finding strategy in a UEFI executable. The UEFI standard makes clear that all memory is identity mapped and the Application Processor (AP) is executing in EL1 [17], allowing non-cached physical access after setting the appropriate system registers.

Our strategy first zeros a large area of memory. Then, we iterate through it and, at each iteration step, check if the value read is zero; if it is, we proceed to write the address of that memory location at that location, and continue iterating. If we encounter any non-zero values, the value read will contain the address of the aliased location. Our alias discovery algorithm is presented in Appendix C. This search strategy allows accurate alias detection in a single memory sweep and significantly improves on previous work [15], enabling us to find physical memory aliases on ARM platforms with SPD modified DIMMs installed.

Aliasing Semantics On our evaluation platforms, we found that the DRAM address space is split into two halves such that the first and second halves alias with each other; cf. Figure 1. This is the case across several DDR4 ARMv8-A platforms (see Table 1), with some of these platforms having multiple DIMMs installed. This indicates that, unlike some other platforms [15], there is no memory scrambling used. As aliasing occurs in DRAM address space, it must be mapped onto physical address space to be useful. Consider the example of the Morello SDP, which has two relevant physical regions that map to DRAM addresses:

- `0x080000000` - `0x0100000000`, "DRAM0"
- `0x8080000000` - `0x10000000000`, "DRAM1"

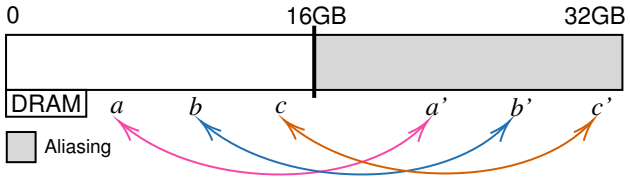


Figure 1: Aliasing as viewed from DRAM address space. Aliasing starts at the 16GB boundary. Addresses closer to 0 have their alias earlier in the second region; a and a' alias, as do b and b', etc.

For this platform, with two 8GB DIMMs installed, the available memory has an apparent capacity of 32GB but a true capacity of 16GB. This presents itself as the second 16GB of memory being an alias to the first 16GB, and vice-versa. In this setup, we can find the corresponding alias for a physical address by adding 16GB to it. If the result of this lies between the DRAM0 and DRAM1 regions, i.e. not within a region of physical memory that is mapped to DIMMs, we can calculate the offset. We add the difference between the start of DRAM1 and end of DRAM0 to get an in-range physical address (see Figure 2). In this example, the alias of address `0x080000000` is `0x8400000000`. This is because DRAM addresses are mapped linearly into physical address space on all of our test platforms. While each of our test platforms have different memory maps, this behaviour is consistent across all of them, and these differences are easily accounted for.

4.2 Userspace Exploit Methodology

In order to make use of this memory aliasing from userspace, the attacker's end-goal is to control memory that aliases to one of their own L3 page tables. Although PTE-spraying alone would quickly cause the system to crash under aliasing, it can be combined with additional coarse allocation manipulation to prevent this. This combination of PTE-spraying with coarse allocation manipulation forms the *alias-tolerant* memory massaging strategy that enables DISARMED. After sufficient spraying the attacker searches for one of its mappings that has a usable alias to a page descriptor. Then they modify the page descriptor so that it maps a desired page in

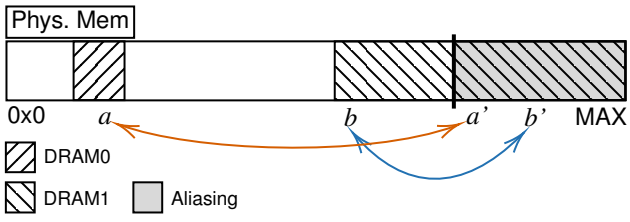


Figure 2: Aliasing as viewed from physical address space. In physical address space, address ranges that are not mapped to DRAM *stretch* aliases.

physical memory into the attacker's address space, acting as a physical-address read/write primitive. Here we describe the configuration with no physical address randomization; the effect of randomization is explored in Section 5.

Exploiting Huge Pages To prevent system crashes, an attacker can use HugeTLB mappings to coarsely control how pages are allocated spatially in physical memory, avoiding aliasing-induced system instability. As the kernel allocates specific regions of memory at system startup for its own use, there are certain regions that the attacker must avoid overwriting when doing page spraying. An example is shown in Figure 3. On all of our evaluation devices, the kernel data and the kernel code regions reside in lower physical memory. There are also reserved memory ranges at the very 'high' end of physical memory, growing down from the highest possible physical address; these are used by the kernel for its own runtime data. Consequently, there is a region in the 'middle' of the 32 GB physical memory layout that contains aliases to this critical kernel runtime data, as shown in Figure 4.

There is a memory region corresponding to the reserved high memory range just before the aliasing boundary, and a region corresponding to the kernel code and data after the alias boundary. This middle region is *not* reserved in the allocator's view—it appears as unused memory—and so if it is written to, critical kernel data will be damaged. An attack must therefore avoid affecting this middle region.

When allocating large amounts of memory (e.g., when page-spraying), the kernel page allocator generally allocates memory in `ZONE_NORMAL` from the lowest physical memory address range growing upwards. As such, if the attacker only makes typical allocations, this middle region is overwritten before their process is allocated useful aliased pages. In order to map two physical addresses that alias to each other, they must find a method to 'jump over' this middle region.

Huge pages are used to perform this 'jump': the pool of reserved memory used for 1GB HugeTLB pages is reserved at the *top* of memory at boot time. During kernel initialization, the HugeTLB reservation protects that memory region from being used by other processes. It also is no longer considered part of `ZONE_NORMAL` for the purposes of memory allocation, decreasing the zone's overall size. However, a process wanting to use the reserved huge pages can request them via an `mmap` call with the `MAP_HUGETLB` flag, which returns a mapping corresponding to this region at the top of the memory. This means the attacker can use `mmap` with the `MAP_HUGETLB` and `MAP_POPULATE` flags in their userspace process to ensure they have a range of high physical memory addresses under their control; see Figure 5.

Allocation Setup The attacker can now use the described properties of hugepage allocation to start the DISARMED attack (see Listing 2). A `hugepage` allocation is prepared with the `mmap` call; after the initial allocation, the attacker writes

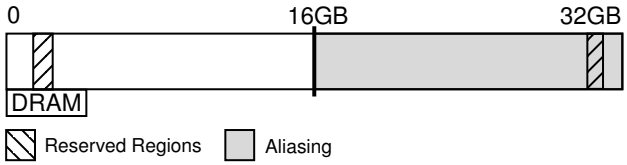


Figure 3: Contents of DRAM, with regions reserved for kernel use

the address of each 64-bit aligned entry to itself *i.e.*, `*entry = (uint64_t)entry` . This serves two purposes: First, it allows for these regions to be trivially checked for changes later; the attacker can compare an address with the data stored at that address to determine if it has been changed. Second, it forces the kernel to map these pages into physical memory at this time so that they can be overwritten later, avoiding issues with CoW semantics. With the huge pages prepared at the top of memory, the attacker can try to overwrite their contents with aliased PTEs. The next stage is to spray and modify page tables to remap physical memory into attacker-controlled address space.

Spraying Strategy The attacker sprays a large number of process page tables into memory, until the majority of non-aliased memory is exhausted. They do this by a creating an anonymous file with `memfd_create` and filling it with a known guard value; this will be used to detect modification later. Repeated calls to `mmap` with the `MAP_SHARED` flag causes the file to stay resident in memory once but appear in their virtual address space many times. In order to facilitate this, the kernel allocates increasingly more page tables to describe this mapping.

By carefully choosing the file size and the number of times they `mmap` this file, the attacker is able to fill most of non-aliased memory with page tables for their process. This results in a majority of these tables being level 3 tables that contain only valid page descriptors, which can be modified to the attacker's advantage without any additional knowledge of physical memory layout.

Finding Aliases The attacker can now check on their previously allocated `hugepages` (see Figure 6). This is done by iterating through the hugepage mapping; for every 8-byte aligned address, compare the address to its contents. These were previously set to be equal for each address; as such, a discrepancy here indicates that some clobbering has occurred due to memory aliasing.

In this case, this data has been overwritten by page table entries after spraying. As the attacker has released no memory, all page tables constructed for the process are still valid. Therefore regions of memory that have been overwritten alias to currently live page tables for this process.

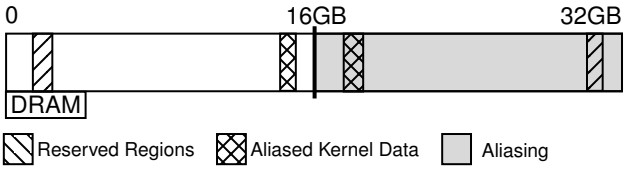


Figure 4: Contents of DRAM with aliasing; kernel regions are now aliased into the *middle* of DRAM

Page Table Modification The attacker, with aliases to live page tables for their process, can now remap physical memory into their address space. It is sufficient to use this to modify a single page descriptor in a level 3 table; this has the effect of remapping a single page of their virtual address space to a location of their choosing in the physical address space. The attacker discovers which page the discovered entry maps by scanning for discrepancies (Listing 3).

TLB Eviction It is essential to evict the modified page descriptor from the TLB. As the page descriptor is modified through memory aliasing, the architecture will not notice the changes. This introduces a cache incoherency problem; repeated accesses of the same virtual page will result in the now stale entry in the TLB being used rather than the modified value. To overcome this, the attacker must make sure this page descriptor is evicted from the TLB whenever it is modified. They iterate over a section of their mapped memory and read each value. This causes enough address translations to occur that older entries—such as the modified page descriptor—are evicted to make space. On the next access that uses the modified page descriptor, the attacker's modified value is used.

Read/Write Anywhere With the ability to modify their own page tables, the attacker's last step is to find the virtual page that this page descriptor maps to. After modifying the page descriptor, they scan through all copies of their sprayed file; one of these pages will no longer contain the guard value. This must be the page that is linked to their chosen page descriptor. They can now remap this page anywhere in physical memory by modifying this chosen page descriptor and read-/write to it as expected. When their process exits, the Linux kernel will notice this stray page descriptor during cleanup; to avoid this, they can simply set it back to its original value.

5 Evaluation on Real Hardware

We evaluate DISARMED on our three ARMv8 platforms (Table 1) using our implementation (Section 4).

Root Escalation We were able to perform attacks against the kernel on all three platforms by patching `modprobe_path`, enabling escalation from an unprivileged user to root.

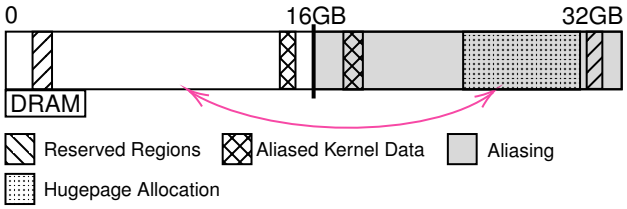


Figure 5: Contents of DRAM with `hugepages` allocated; they reside towards the top of DRAM.

PKCS#11 Key Extraction We additionally perform successful TrustZone attacks against two of our evaluation platforms, the Morello SDP and the LS1046ARDB. This is done after examining publicly available information, such as the the ARM TF-A source code and respective developer references, to determine the layout of the physical memory and the location of secure memory. Using these details we deploy DISARMED and obtain the entire contents of the secure memory, consisting of the Secure Monitor, TEE, and trusted applets, including all secret data. To verify that DISARMED is able to retrieve secret data from the secure world, we attack the OP-TEE PKCS#11 driver used by OpenSC's `pkcs11-tool`. OP-TEE is a common TEE implementation built on TrustZone [50], and PKCS#11 is a standardized API for interacting with cryptographic tokens [36]. The `pkcs11-tool` utility uses this API to interact with smart cards, focussing on those used for authentication, encryption, and digital signatures. As an alternative to a physical smart card, the OP-TEE PKCS#11 driver allows users to store secrets in the secure memory region. By reading the contents of secure memory using DISARMED, a userspace program is able to recover material such as RSA-4096 private keys that are ordinarily shielded from retrieval by the TEE. This demonstrates practical secret retrieval against a standardized secure key storage solution built upon TrustZone. The code to perform this recovery is in Appendix B.

Configuration These attacks use a 1GB HugeTLB page configuration reserved at boot time, with a variable number of huge page reservations yielding the same results. 2MB-

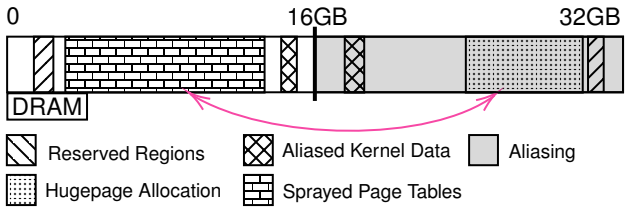


Figure 6: Contents of DRAM with `hugepages` and spraying; aliases now exist between sprayed page tables and the hugepage allocation.

sized HugeTLB pages, THPs, or non-reserved 1GB HugeTLB `hugepages` do not yield the needed allocation behaviour from the kernel. In particular, reserving an equivalent amount of 2MB HugeTLB `hugepages` results in a reserved region at the lower end of memory. With 1GB `hugepages`, page tables typically appear at the start of the huge page allocation. Tests indicate that a L3 page table is the first page of the hugepage allocation after spraying on our test platforms.

Physical KASLR We observe that the Ampere Altra DP is configured to use KASLR by default, including support for physical KASLR. In many configurations, even if KASLR is enabled, the kernel will only randomize its location in the virtual address space. We find that virtual KASLR had no impact on our attack's success. However on arm64 platforms, when a KASLR-capable Linux kernel is booted from a UEFI implementation that supports `EFI_RNG_PROTOCOL` via the EFI stub boot-loader, it will additionally randomize its location in physical memory [49]. While our attack is eventually successful with physical KASLR, this presents two challenges:

1. *Time*: With the kernel location in physical memory now unknown, far more memory must be searched for our victim. Ideally an attacker would be able to locate the kernel by, for example, examining `TTBR1_EL1`, but this is not possible from userspace. Instead the attacker must scan for their victim in memory.

2. *System Stability*: As described, the kernel typically resides low in memory. With physical KASLR enabled, it can reside anywhere in memory. If this is the upper half, it is likely page table spraying will overwrite it too early, causing a crash. As such the success of the attack is dependent on the random location in memory. This means that an attack can require multiple reboots of the target system.

System Stability At runtime, we observe that our aliased test platforms remain stable under low memory pressure. As memory pressure increases, the kernel eventually begins to hand out aliases of in-use pages; this causes a number of stability issues, such as kernel panics and deadlocks. The kernel's preference to hand out lower addressed pages and reuse of reclaimed pages allows an aliased system to remain operational for a long time as long as memory pressure remains low. We also occasionally observed various data corruption issues that persisted across boots e.g., silent database corruption and corrupted `inodes`. These are recoverable via specific targeted action (e.g., `fsck`), which is done with normal unaliased DIMMs to prevent further long-term issues.

On platforms with physical KASLR disabled—including the Ampere Altra DP with KASLR disabled—both the kernel and TrustZone attacks are able to reliably modify memory without system crashes. On one platform, the Ampere Altra DP, we experience persistent issues with firmware and possible NVRAM corruption.

6 Mitigations

We propose two mitigations for DISARMED. One seeks to detect aliasing at boot, and the other aims to thwart *alias-tolerant* memory massaging techniques.

6.1 Aliasing Detection

Our first proposed mitigation strategy is a detection mechanism implemented in the device firmware. This strategy detects aliasing by first writing a known magic value to a page-aligned memory address, and then scanning all page-aligned memory for this value (See Listing 4). If it is read at any address that is not the one it was written to, this indicates that there is some memory-aliasing at play, and the firmware panics. It is essential to choose a value that otherwise is not present in memory at boot to prevent false-positives from rendering the system inoperable.

To evaluate the performance impact of our mitigation, we implement it in TF-A on the NXP LS1046ARDB platform in the BL2 firmware stage. With this we are able to scan all of the relevant page-aligned addresses in about $\sim$1 second with 32GB of memory installed, representing a minimal impact on boot time. This represents a reliable way to detect dangerous aliasing schemes that are present at boot-time, such as SPD modification. This mitigation can detect any aliasing scheme that is less fine-grained than the page level; while this mitigation is appropriate for our test platforms, it may be the case that page-unaligned aliasing is possible on some other platforms. In these cases, this mitigation may need platform-specific adaptations.

This mitigation is also suitable to be implemented elsewhere in the trusted boot chain, such as the boot-loader or kernel, with the only requirement being that it needs to run before untrusted code has executed. Our example implementation of this mitigation, used here for performance evaluation, is available as an artifact in Appendix B. It may be possible to bypass this mitigation with a BatteringRAM-style interposer [16], where the aliasing can be dynamically toggled, preventing boot-time detection.

6.2 Deterministic Page Allocation

Our *alias-tolerant* memory massaging strategy relies on some determinism from the kernel page allocator. As such, introducing non-determinism into the page allocator can act to hinder an attacker's ability to make use of aliasing. One mechanism for this in Linux, disabled by default, is `CONFIG_SHUFFLE_PAGE_ALLOCATOR`. This impacts potential massaging techniques in two ways:

1. *Reduced stability*: Much like physical KASLR reduces an attacker's ability to reason about physical memory layout (and therefore avoid clobbering kernel data), `SHUFFLE_PAGE_ALLOCATOR` also prevents an attacker from reliably avoiding damaging kernel data.

2. *Spraying non-determinism*: The attacker is no longer able to predict how much spraying is suitable for the current memory configuration and pressure.

We find that `SHUFFLE_PAGE_ALLOCATOR` mitigates our described memory massaging strategy. Targeting memory massaging techniques is less robust as new strategies can be found, but is suitable for a defence-in-depth approach.

7 Discussion

7.1 Implications For Further Attacks

We previously define alias-tolerant memory massaging as a required primitive for DISARMED in Section 3.2. We believe it is feasible to adapt other existing memory massaging techniques to be alias-tolerant. As an example, we document a potential alternative attack strategy below.

Page Re-use Attacks We explored an attack strategy based around page reuse, which would avoid needing to perform the *jump* we describe with `hugepages`. Through allocating a large amount of memory and slowly populating it, it is possible to observe some physical memory aliasing in process address space. While the Linux page allocator typically prefers to allocate memory linearly, it was sometimes possible to be allocated a very small number of physically aliased pages before some critical failure due to damaging kernel data. We believe that this could be combined with a Frame Feng Shui style attack [28] to have a physical page reused while the attacker controls its alias. However, we found that the system was in a very fragile state when these aliases were discovered; typically much more allocation would overwrite some critical data leading to a crash or hang. Additionally, it would no longer be possible to target page tables due to their differing Migratetype compared to the mapped memory [12]. Current Migratetype escalation strategies require exhausting most of available physical memory, which is not possible under memory aliasing conditions without clobbering critical data.

While this was ultimately not pursued due to issues with system stability, it may present an alternative exploitation strategy if some more reliable strategy was found; this would free our current implementation of DISARMED of the need for hugepage allocation.

x86 Userspace Aliasing We believe that the strategies that we describe as part of DISARMED do not have to be architecture dependent. Page tables are a frequent target for manipulation, as doing so can grant an attacker control over physical memory mappings. As such, attacks that make use of page table manipulation are well documented across multiple architectures [25, 51, 57], and we have no reason to believe that our proposed *alias-tolerant* memory massaging strategy is ARM-specific. It *is* dependent on the memory layout of the target, although there are no issues with the differences in the layouts across our test platforms.

Prior work [15] has demonstrated that the ability to read and write physical memory in combination with aliasing is sufficient to break AMD SEV-SNP. In combination with the strategies we present here to obtain such a physical memory read-write mechanism, it may be feasible that DISARMED can be applied for userspace aliasing attacks on other architectures and TEEs, such as AMD SEV-SNP.

7.2 Analysis of ARM Security Measures

Here we perform a brief analysis of other relevant privilege separation mechanism for the ARM architecture, and how they may be affected by memory aliasing attacks.

ARM Confidential Compute Architecture (CCA) is a natural extension of the TrustZone architecture to permit non-secure code to manage *Realms* [8]. However, unlike TrustZone, untrusted external memory (*i.e.*, DRAM) is considered in the CCA threat model, including physical attacks against DRAM [5]. In accordance with this, memory encryption is used [5, 8], and some critical components are either stored in on-chip memory [5] or in external memory with some integrity checks [5]. This is an improvement over TrustZone, although prior analysis indicates that the recommended cryptographic schemes do not provide integrity or freshness guarantees, which may enable some memory aliasing attacks [15].

CHERI is an extension of existing RISC Instruction-Set Architectures (ISAs) with *architectural capabilities* [53]. Such protection can be used to implement spatial memory safety [54], compartmentalisation primitives [55], and temporal memory safety [56]. The CHERI protection model leans heavily on trusted DRAM; the CHERI adversary model explicitly excluding hardware attackers. CHERI stores capability validity bits out of band, either in some of the ECC bits or in an unaddressable region of DRAM. In this second case, memory aliasing could allow access to the entire validity bit store, which would allow capabilities to be forged. Additionally, it is likely possible to forge a capability simply by modifying an existing capability via its alias. Such capability forgery, if made possible by memory aliasing, would defeat the CHERI security mechanism.

7.3 Targeting Other Platform Types

Knowing that memory aliasing attacks are feasible on ARM architectures opens the gates for attacks targeting other types of platforms, such as laptops and phones. Our test platforms are intended for use as servers or workstations, where memory modules are installed via DIMM slots. Smaller form factors often do not use DIMMs due to design constraints. Additionally, mechanisms governing memory configuration vary between platform types. These systems will determine memory configuration through other means, such as:

1. *Resistor straps*: Some platforms use *straps* to determine memory capacity. Resistors (*i.e.*, straps) pull pins high or low to configure certain SoC options.
2. *Stored in BIOS*: Instead of being stored on an dedicated chip, SPD data can be stored in the BIOS image.
3. *Read directly from DRAM registers*: A SoC may read density directly from the (typically read-only) mode registers of a DRAM chip.

We distinguish two broad approaches to inducing BadRAM-style static aliasing, and the typical DRAM formats for different form factors.

Hardware modification As current methods of inducing aliasing require physical access, we consider portable computing platforms a natural target for memory aliasing exploits. A typical memory module package used on some laptops are SO-DIMMs, which are smaller than regular DIMMs and as such well-suited for laptop form factors. SO-DIMMs also use EEPROMs storing SPD data to store configuration details, and thus are open to potential modification. Some laptop models do not use DIMM slots, instead using DRAM chips soldered directly to the motherboard. While it may be possible to cause aliasing by replacing these chips with lower capacity memory chips, this increases exploit complexity and difficulty considerably compared to SPD modification.

Targeting memory configuration instead of replacing memory modules may provide an easier attack vector. Laptops using soldered-on memory typically will use straps, especially models with multiple variants available with distinct memory configurations. With knowledge of how straps correspond to memory capacity, an attacker may be able to remove and place resistors to adjust the reported memory capacity. Whether straps are used for memory configuration, and what layouts correspond to which capacities are however not standardised and will be chip-set specific. On platforms where the capacity is determined by reading the DRAM mode registers, any attempts to modify these values would be a significant undertaking requiring specialized engineering skills.

Firmware modification Many laptops and some tablets store SPD data in the BIOS. Depending to what extent BIOS firmware (and the SPD data) is protected from tampering, modifying the firmware itself is a potential target. To support multiple vendors and memory capacities, a typical setup is such that the SPD data contains a lookup table for supported DRAM models, and an entry containing the model name for the installed memory. Without a secure boot flow protecting the SPD data, attackers may be able to change these values as a potential means of inducing aliasing.

8 Conclusion

We presented DISARMED, a new attack that extends memory aliasing attacks into userspace. With one time physical access required to induce aliasing, an attacker is able to construct a physical memory read/write primitive from unprivileged userspace, allowing the full compromise of both the running Linux kernel and ARM TrustZone. We showed and implemented the first *alias-tolerant* memory massaging strategy to achieve this, and demonstrated its efficacy across multiple ARMv8 platforms.

Additionally, we implemented and demonstrated a mitigation for DISARMED, which detects these aliases at boot time with minor performance impact. Finally, we discussed the broader applicability of DISARMED, suggesting other potential *alias-tolerant* memory massaging strategies and the applicability of aliasing attacks against other ARM security mechanisms.

9 Acknowledgments

This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/V000454/1, EP/X03738X/1, and EP/X03738X/2. The results feed into DSbDtech. This work was further supported by Innovate UK and the Department for Science, Innovation and Technology for the adoption and diffusion of CHERI technology under project 10167245 (“CHERI-Zephyr++: A memory-safe, lightweight real-time operating system for critical RISC-V embedded systems”).

A Ethical Considerations

Responsible Disclosure

We disclosed our initial results and the underlying vulnerability to ARM in February 2025. ARM acknowledged our findings but noted that physical attacks are outside the scope of their threat model. They further remarked that:

> "TF-A images are run from either ROM or on-chip trusted SRAM. This means TF-A is not vulnerable to an attacker that can probe or tamper with off-chip memory."

While we confirm that TF-A indeed is able to run from memory-mapped SRAM, in practice it is unusual to run only from SRAM due to space constraints. Indeed, in this paper, we show that a subsequently booted TrustZone OS such as OP-TEE typically uses DRAM and is hence vulnerable to attacks based on memory aliasing despite supporting operation from SRAM [50].

Following this, ARM opened their Trusted Firmware bug bounty program. We resubmitted our work under this program following direct encouragement from ARM; in reply, ARM remarked:

> "At this time, Arm CPUs aren't in scope for either of our bug bounty programs [...] This attack also does not negate the security guarantees provided in Arm CCA or other Arm IP."

Ethical Analysis

There were no experimental participants involved in this work. Additionally, all experiments were carried out on dedicated systems in a lab environment; as such, there was no interference with any live systems. In the context of direct impact, we feel that we have minimised risk and harm by having no experimental participants and additionally minimised harm to wider society by isolating work in a lab environment.

Our work represents a weakening of the security of TrustZone, which is widely deployed and depended upon. Because TrustZone is used to store confidential information, the presentation of an attack risks doing harm to people that rely on TrustZone-based TEEs.

During the process of pursuing this work, we disclosed it to ARM as described previously. We feel that the publication of our work does not immediately expose a large number of systems to this new attack vector. Attacks require physical access, preventing an immediate large-scale attack based on our work. Instead a level of premeditation is required to make use of DISARMED. We also propose a suitable mitigation that can be implemented to mitigate DISARMED that we feel is practical and effective.

We have no reason to believe that this work and our pursuit of it has legal ramifications. We have complied with relevant laws in our jurisdiction, performing this research in a lab environment with no experimental participants. All used documentation is publicly available under appropriate licensing from the equipment manufacturers. However, we acknowledge that TEEs can be employed in the implementation of Digital Rights Management (DRM). As such, any attack against TEEs risks being used by a bad actor to violate intellectual property law by, for example, retrieving encryption keys used in the decoding of DRM protected media in the future.

After considering the ethical implications of this research as described, we felt it appropriate to continue with this work. We concluded that the pursuit of this work represented minimal risks and harms to wider society, ARM, and other stakeholders. Where there was risk, it was appropriately mitigated.

Additionally, we feel that we have performed reasonable diligence before submitting this work for publication. The disclosure of this work to ARM as described, combined with the provided mitigation and non-immediacy of exploitation, are the reasons that we feel it is also appropriate for this work to be made public despite the described risk to users of TrustZone. As memory aliasing attacks are quite young, our hope is that DISARMED can influence future memory protection design for the better while representing a managed risk to existing trusted computation.

B Open Science

Artifact Summary

All of our artifacts are available here: https://github.com/bhamsec/disarmed. We include these artifacts:

- *Attack Implementation*: The implementation of DISARMED that we used to evaluate its efficacy and performance across our test platforms.
- *QEMU Patch*: A patch to QEMU 9.2.3 that allows for the emulation of the memory aliasing behaviour that we observed.
- *UEFI Scanner*: The implementation of the scanning code that we wrote to discover the behaviour of memory aliasing on ARM platforms.
- *Prototype Mitigation*: A patch to ARM TF-A 2.10.0 that we used to evaluate the performance impact of our proposed mitigation.
- *GDB Tooling*: A GDB script developed to assist in the debugging of page table attacks.
- *OP-TEE TA*: A trusted applet used for testing.

- *PKCS#11 Key Extractor*: Code that extracts RSA-4096 private keys from memory dumps made by DisARMed. This allows for the extraction of keys from the OP-TEE PKCS#11 trusted applet.

Other Relevant Code

Some of the code used in our evaluation of DISARMED was not original: specifically, we used the SPD modification code implemented in BadRAM [15]. It can be found here: https://github.com/badramattack/badram/tree/main/scripts/pi-pico/ee1004.

Emulating Aliasing Behaviour

One of our artifacts is a patch to QEMU. Here, we discuss using an emulated environment with similar aliasing behaviour for developing these userspace aliasing attacks. We modify QEMU to replicate the observed aliasing behaviour, enabling easier debugging of these attacks.

This is achieved by creating this memory aliasing in host virtual address space using `MAP_SHARED`. Then, this region is used to back guest physical memory. By inducing aliasing in host virtual address space, aliasing in guest DRAM space is emulated by the host Memory Management Unit (MMU). This allows for performant aliasing emulation as the heavy lifting is done by the host MMU.

We found that this is sufficiently accurate for attack development; we are able to transfer code that targeted this emulated aliasing to bare metal with only platform-specific changes (*i.e.*, memory mapping). We hope that this can be used to make exploring these aliasing-from-userspace attacks more straightforward and accessible.

C Alias Discovery Algorithm

```
void find_aliases(uint64 start, uint64 end) {
  configure_cache();/* disable caching */
  memset(start, 0, end - start); /* clear region. future non-zero reads indicate aliasing */
  for (uint64 addr = start; addr < end; addr += sizeof(uint64)) {
    if (read_u64(addr) != 0)/* aliasing has caused this data to be overwritten */
        printf("found alias! 0x%zx, 0x%zx\n", addr, read_u64(addr));
    *addr = addr; /* write address. if this is found later, aliasing has occurred */
  }
}
```

Listing 1: Simplified alias discovery algorithm. A single sweep of memory is done between addresses `start` and `end` to search for aliases.

D Huge Page Allocation Algorithm

```
  map = mmap(NULL, HUGEMAP_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC,
             MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB | MAP_HUGE_1GB ,-1, 0);
  for (uint64_t* p = (uint64_t*)map; (void*)p < (void*)(map + HUGEMAP_SIZE); p++) {
    *p = (uint64_t)p;
  }
```

Listing 2: Simplified hugepage preparation. Addresses of each mapping are written into their respective locations.

E Modified Pages Algorithm

```
  for (size_t i = 0; i < SHM_COUNT; i++) {
    for (uint64_t* p = maps[i]; (void*)p < (void*)((char*)maps[i] + FILE_SIZE); p += (page_size / sizeof(
    uint64_t))) {
      if (*p != magic) {
        printf("found modified page!\n");
        window = p;
        goto next;
      }
    }
  }
```

Listing 3: Searching for modified pages after page table modification. The lack of the previously written guard value indicates a page has been remapped.

F Mitigation Algorithm

```
int scan_region(void* start, void* end, uint64_t magic) {
  for (uint64_t* test = start; test < end; test += (PAGE_SIZE / sizeof(uint64_t))) {
    if (*test == magic && test != DRAM0_START) {
      return 1;
    }
  }
  return 0;
}

int alias_check(uint64_t magic) {
  *DRAM0_START = magic;
  return scan_region(DRAM0_START, DRAM0_END, magic) || scan_region(DRAM1_START, DRAM1_END, magic);
  return 0;
}
```

Listing 4: Simplified mitigation algorithm. A known magic value is searched for in memory.

References

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3:18–24, 2004.

[2] Amazon. Amazon Web Services: Graviton Processors. https://aws.amazon.com/ec2/graviton/.

[3] ARM. CoreLink TrustZone address space controller TZC-380 technical reference manual, revision: r0p1. https://documentation-service.arm.com/static/5e8e28cffd977155116a6798, 2010.

[4] ARM. Learn the architecture - AArch64 memory management examples, June 2019.

[5] ARM. Arm CCA security model 1.0, 2021.

[6] ARM. Arm architecture reference manual for A-profile architecture, 2022.

[7] ARM. Arm Morello system development platform technical reference manual. https://developer.arm.com/documentation/102278/latest, 2022.

[8] ARM. Learn the architecture - introducing Arm confidential compute architecture, 2023.

[9] Gal Beniamini. Full TrustZone exploit for MSM8974. https://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html, August 2015.

[10] Gal Beniamini. QSEE privilege escalation vulnerability and exploit. https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, June 2016.

[11] Gal Beniamini. Project Zero: Trust Issues: Exploiting TrustZone TEEs, July 2017.

[12] Matej Bölcske, Patrick Jattke, Johannes Wikner, and Kaveh Razavi. Rubicon: Precise microarchitectural attacks with page-granular massaging. In *10th IEEE European Symposium on Security and Privacy (S&P)*, June 2025.

[13] David Buchanan. Can you get root with only a cigarette lighter? https://www.da.vidbuchanan.co.uk/blog/dram-emfi.html, 2024.

[14] Liang Cai. Guard Your Data with the Qualcommm Snapdragon Mobile Platform: Hardware-backed Mobile Secure Storage. Technical report, Qualcomm Technologies, Inc., April 2019.

[15] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In *46th IEEE Symposium on Security and Privacy (S&P)*, May 2025.

[16] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. Battering RAM: Low-cost interposer attacks on confidential computing via dynamic memory aliasing. In *47th IEEE Symposium on Security and Privacy (S&P)*, May 2026.

[17] UEFI Forum. UEFI specification 2.10. https://uefi.org/specs/UEFI/2.10/, 2022.

[18] GlobalPlatform. GPD_SPE_021, GlobalPlatform device committee TEE protection profile version 1.0, 2013.

[19] Mathieu Gross, Nisha Jacob, Andreas Zankl, and Georg Sigl. Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC. *Journal of Cryptographic Engineering*, 12(2):181–196, June 2022.

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 300–321. Springer International Publishing.

[21] Joffrey Guilbon. Attacking the ARM's TrustZone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html, July 2018.

[22] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[23] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, New York, NY, USA, 2017. Association for Computing Machinery.

[24] JEDEC. SPD annex L: Serial Presence Detect (SPD) for DDR4 SDRAM modules, release 3, 2015.

[25] Jonas Juffinger, Sudheendra Raghav Neela, Martin Heckel, Lukas Schwarz, Florian Adamsky, and Daniel Gruss. *Presshammer: Rowhammer and Rowpress Without Physical Address Information*, page 460–479. Springer Nature Switzerland, 2024.

[26] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 957–972. USENIX Association, 2014.

[27] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. 42(3):361–372.

[28] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, page 695–711. IEEE, May 2020.

[29] Arm Limited. Trusted Firmware-A documentation. https://trustedfirmware-a.readthedocs.io/en/latest/index.html, 2025.

[30] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1963–1976, 2022.

[31] Antonis Louka, Jesse De Meulemeester, Steven Keuchel, Ingrid Verbauwhede, and Jo Van Bulck. Physical memory please: Practical memory-aliasing attacks on risc-v pmp. *Proceedings of the Microarchitecture Security Conference*, February 2026. ISSN 3052-2749. doi: 10.46586/uasc.2026.008. URL http://dx.doi.org/10.46586/uasc.2026.008.

[32] Microsoft. Google Cloud: Arm VMs on Compute. https://docs.cloud.google.com/compute/docs/instances/arm-on-compute, .

[33] Microsoft. Microsoft Azure pricing: Virtual Machine series. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/series/, .

[34] Microsoft. Microsoft Azure: B family general purpose VM size series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/b-family, .

[35] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[36] OASIS Open. PKCS #11 cryptographic token interface base specification version 2.40. https://optee.readthedocs.io/en/latest/index.html, 2015.

[37] Oracle. Oracle Cloud Infrastructure: Arm-Based Compute. https://www.oracle.com/cloud/compute/arm://www.oracle.com/cloud/compute/arm/.

[38] Sam Page. Kernel exploitation techniques: modprobe_path. https://sam4k.com/like-techniques-modprobe_path/, 2022.

[39] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 195–209. ACM, November 2019.

[40] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*, August 2016.

[41] Mark Seaborn and Thomas Dullien. Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges. URL https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[42] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of Security-Oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, August 2017. USENIX Association.

[43] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, July 2018.

[44] The Kernel Development Community. pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt, 2025.

[45] The Kernel Development Community. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html, 2025. [Version 6.16.0].

[46] The Kernel Development Community. HugeTLB Pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html#hugetlb-pages, 2025. [Version 6.16.0].

[47] The Kernel Development Community. Physical Memory. https://docs.kernel.org/mm/physical_memory.html, 2025. [Version 6.16.0].

[48] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698.

[49] Linus Torvalds. Linux, `arch/arm64/Kconfig`. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, 2025. [Version 6.15.3].

[50] TrustedFirmware.org. OP-TEE documentation. https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html, 2023.

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16. ACM, October 2016.

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 92–113. Springer International Publishing, 2018.

[53] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019.

[54] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020.

[55] Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, Sep 2016.

[56] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020.

[57] Nicolas Wu. Dirty Pagetable: A novel exploitation technique to rule linux kernel. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html, 2023.

[58] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41.