Preventing Kernel Hacks with HAKC

Derrick McKee*, Yianni Giannaris†, Carolina Ortega Perez†, Howard Shrobe†, Mathias Payer‡,
Hamed Okhravi§, and Nathan Burow§
*Purdue University, †MIT CSAIL, ‡EPFL, §MIT Lincoln Laboratory

***Abstract*—Commodity operating system kernels remain monolithic for practical and historical reasons. All kernel code shares a single address space, executes with elevated processor privileges, and has largely unhindered access to all data, including data irrelevant to the completion of a specific task. Applying the principle of least privilege, which limits available resources only to those needed to perform a particular task, to compartmentalize the kernel would realize major security gains, similar to microkernels yet without the major redesign effort. Here, we introduce a compartmentalization design, called a *Hardware-Assisted Kernel Compartmentalization* (HAKC), that approximates least privilege separation, while minimizing both developer effort and performance overhead. HAKC divides code and data into separate partitions, and specifies an access policy for each partition. Data is owned by a single partition, and a partition's access-control policy is enforced at runtime, preventing unauthorized data access. When a partition needs to transfer control flow to outside itself, data ownership is transferred to the target, and transferred back upon return. The HAKC design allows for isolating code and data from the rest of the kernel, without utilizing any additional Trusted Computing Base while compartmentalized code is executing. Instead, HAKC relies on hardware for enforcement.**

Loadable kernel modules (LKMs), which dynamically load kernel code and data providing specialized functionality, are the single largest part of the Linux source base. Unfortunately, their collective size and complexity makes LKMs the cause of the majority of CVEs issued for the Linux kernel. The combination of a large attack surface in kernel modules, and the monolithic design of the Linux kernel, make LKMs ideal candidates for compartmentalization. To demonstrate the effectiveness of our approach, we implement HAKC in Linux v5.10 using extensions to the Arm v8.5-A ISA, and compartmentalize the `ipv6.ko` LKM, which consists of over 55k LOC. The average overhead measured in `Apachebench` tests was just 1.6%–24%. Additionally, we compartmentalize the `nf_tables.ko` packet filtering LKM, and measure the combined impact of using both LKMs. We find a reasonable linear growth in overhead when both compartmentalized LKMs are used. Finally, we measure no significant difference in performance when using the compartmentalized `ipv6.ko` LKM over the unmodified LKM during real-world web browsing experiments on the Alexa Top 50 websites.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24026
www.ndss-symposium.org

I. Introduction

Modern kernels have expanded their functionality well beyond the "three easy pieces" of concurrency, virtualization, and persistence [5]. Users expect additional features, such as protocol implementations, advanced filesystems, and driver support for an ever growing number of devices. However, as the kernel has grown, its monolithic design, which provides only a single address space for all kernel functionalities, persists. With the increased size and complexity of the kernel, the number of CVEs issued for Linux per year has grown by over 270% from 2005 to 2020.

Loadable kernel modules (LKMs), which are the mechanisms through which additional functionality gets added, provide a natural compartmentalization boundary for the kernel. As kernel module code can always be compiled into the main kernel image, at the source level, there is no clear difference between core kernel code and LKM code. However, when compiled as individualized units, LKMs are not part of the core kernel image that gets loaded by the bootloader. Instead, the kernel dynamically loads an LKM when the kernel needs the particular functionality offered by the LKM. Thus, at runtime, there is a clear and logical separation between LKMs and the rest of the kernel, but the monolithic design of the kernel effectively erases that separation [69].

Bugs in LKMs become just as severe as other kernel bugs, as all code and data exist in the same address space, with no isolation and executing with elevated privilege. Unfortunately, the sheer size and complexity of LKM code create an attack surface much larger than the core kernel. Of the 567 high severity CVEs we analyzed (see § II-B), 301 were found in the `drivers/` and `sound/` directories (or contained the word "driver" in the CVE description for cases of proprietary code, such as the Nvidia GPU driver). We argue that most of the code in those directories are intended for LKMs, and thus that most high severity CVEs come from LKMs, despite our underapproximation of CVE sources.

As an example of a high severity Linux CVE, consider CVE-2016-4997 [65] listed in Listing 1. The code is part of the IPv4 packet filtering subsystem, and is executed during error cleanup. The exploit involves the attacker supplying a small positive integer value via a system call, which, due to the LKM only performing an upper bound test and not a lower bound test, can lead to a corruption of a structure submember used as an offset value in a pointer computation. The pointer, computed using the offset submember corrupted by the user, is then written to the `me` pointer in line 7, which decrements an underlying integer, allowing an arbitrary kernel integer (e.g., the current process `UID`) to be decremented.

This exploit is an example of a data-only attack that allows for accessing data beyond what the developer intended, which

```
1 static void compat_release_entry(struct
      compat_ipt_entry *e) {
2   struct xt_entry_target *t;
3   struct xt_entry_match *ematch;
4
5   /* Cleanup all matches */
6   xt_ematch_foreach(ematch, e)
7     module_put(ematch->u.kernel.match->me);
8   t = compat_ipt_get_target(e);
9   module_put(t->u.kernel.target->me);
10 }
```

Listing 1: Packet filter code that allows root access. A user-controlled pointer value can be passed to `module_put` in line 7 without violating memory safety or control-flow integrity, and an underlying integer is decremented.

the monolithic design of the Linux kernel happily allows. A properly executed exploitation of this CVE does not violate memory safety, as all memory accesses are in validly allocated and live memory regions, and no practical memory safety mechanisms [1] prevent submember corruption, of which this exploit takes advantage. Control-flow integrity [27] is not violated either, because execution flows along a valid path at all times. Consequently, while existing mitigations such as memory safety and control-flow integrity have a place in securing the kernel, compartmentalization is necessary for truly secure kernels. Compartmentalizing the packet filtering functionality so that accessible memory is restricted to only that which the developer intends prevents such data-only exploits, even in the presence of buggy code. In that way, compartmentalization provides similar security guarantees to microkernels, but without the significant engineering changes that microkernels impose.

Current state-of-the-art commodity kernel protections (as opposed to embedded kernel defenses [13], [95]) generally fall into one of three categories: virtualization-based, microkernel-based, or compiler-based. Virtualization-based protections [73], [79], [86], [87] employ a hypervisor to monitor execution or provide stronger isolation between execution domains. Microkernel-based protections [32], [37] completely redesign the operating system to minimize the Trusted Computing Base (TCB) to typically include only the virtual memory management and IPC, and isolate other traditional kernel services as user-space processes. Compiler-based protections [1], [16], [27] introduce security checks or randomization [68] by the compiler that attempts to thwart code-reuse or data-only attacks. Virtualization and microkernel defenses provide the strongest protections, but are the least performant, and still rely on additional software TCB. Compiler protections, with the exception of KASAN [1], are more performant, but only protect a subset of the attack surface, as with kCFI [27], or are often circumvented [10].

In this paper, we present Hardware-Assisted Kernel Compartmentalization (HAKC), a mechanism for compartmentalizing kernel code and data. HAKC relies on hardware features for enforcement, which avoids growing the TCB, yet provides strong data and control-flow protection. HAKC splits code and data into partitions that contain a developer-specified mix of both, and collects the partitions into a larger grouping for efficient policy enforcement. A data-access policy is specified for each partition within the larger group, and a control policy is defined for the larger group if control flow needs to exit its constituent partition set. HAKC provides fine-grained data-access and control-transition policies to prevent arbitrary data access and code execution. The data-access policy ensures that all data belongs to exactly one partition, and the accessed data conforms to the data-access policy defined for each partition. The control-transition policy checks that indirect control flow targets also conform to the partition set access policy. When control flow exits the partition set, data ownership is transferred to the target, and then restored upon return. In this way, HAKC optimizes and enforces safe local data access; code and data access within a partition is secure and quick, relative to those outside. However, data and code defined outside the partition is accessible, but only if explicitly needed. While we designed HAKC around compartmentalizing LKMs, it is not limited to only that use case; HAKC can be applied to core kernel code, as well as user-space code.

We implement Hardware-Assisted Kernel Compartmentalization for the `ipv6.ko` and `nf_tables.ko` LKMs in Linux 5.10 using hardware features present in the ARMv8.5-A ISA. We measure the performance overhead of compartmentalizing `ipv6.ko` using microbenchmarks, and find that HAKC imposes an average 1.6%–24% overhead. Additionally, we measure the overhead of using two compartmentalized LKMs together, and find that the overhead grows linearly. Finally, when simulating typical browsing behavior using the Alexa Top websites, we find no significant difference using our compartmentalized LKM over an unmodified LKM. To summarize, this paper provides the following contributions:

- A compartmentalization policy API for defining fine-grained compartmentalization policies.
- A practical, hardware-based compartmentalization enforcement mechanism.
- An implementation of a compartmentalization policy on the `ipv6.ko` and `nf_tables.ko` LKMs[1].
- An extensive evaluation on the overhead imposed by our compartmentalization policy and enforcement, demonstrating its practicality.

II. Background and Motivation

Here, we present some background about Pointer Authentication (PAC) and Memory Tagging Extension (MTE), the hardware security primitives we used to build our prototype HAKC implementation. Both PAC and MTE are present in the ARMv8.5-A ISA. We additionally provide an analysis of high severity CVEs that motivate the need for HAKC.

A. Hardware Primitives

Pointer Authentication: Introduced in ARMv8.3, Pointer Authentication is used to cryptographically sign pointers, and store the signature in the "unused" upper bits of a 64-bit pointer (see Figure 1).

PAC implements two instruction classes, one for signing and one for authenticating a signed pointer, and allows for using five different keys, two for data and code pointers each

[1] Available at https://github.com/mit-ll/HAKC

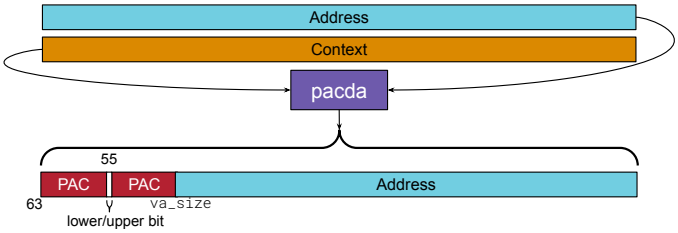


Fig. 1: Pointer signing using PAC. The upper/lower bit indicates if higher bits are used in the PAC signature.

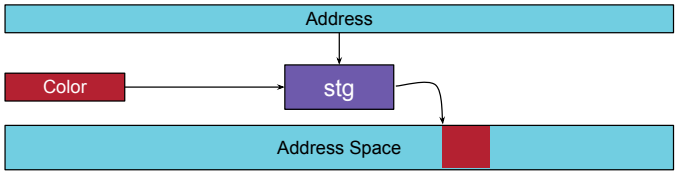


Fig. 2: Address space coloring using MTE.

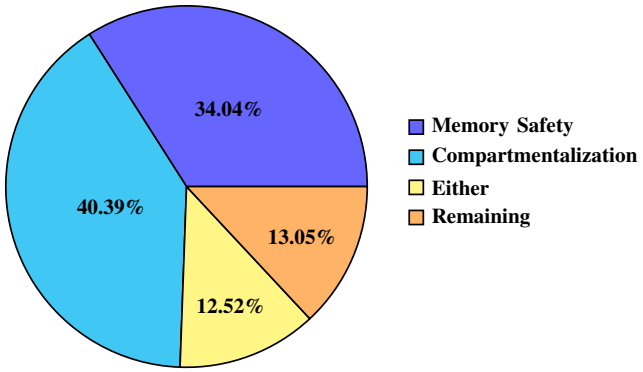


Fig. 3: A breakdown of mitigations for Linux kernel high severity CVEs. "Either" indicates that if memory safety *or* compartmentalization were present, the bug would not be exploitable. "Remaining" indicates that memory safety or compartmentalization would not mitigate the CVE.

and one user-specified key. For example, in Figure 1, the `pacda` instruction specifies using the `a` key for signing `d`ata pointers. Signing involves specifying a pointer to be signed, the key to use for signing, and a 64-bit *signing context*. PAC was initially designed to mitigate code reuse and pointer substitution attacks [3], because the signed pointer no longer references validly mapped memory, and an invalidly modified pointer will fail future authentication. The attacker, therefore, will have to guess a valid signature for a replacement pointer, which is hard because the signature uses the cryptographically secure QARMA block cipher [6]. However, separate address space domains can be established by varying the signing context, because it can be any 64-bit value. For instance, the stack pointer can be used as the context to ensure stack-based buffer overflows do not overwrite valid return addresses with attacker controlled values. Liljestrand et al., implemented a type safety mechanism by using an object ID as the PAC context [50], and Farkhani et al., implemented a temporal memory safety mechanism using allocated object metadata as the context [23]. Other uses for PAC have been proposed [20], [48], and HAKC uses PAC (combined with MTE) to enforce compartments' access policies.

To obtain a valid pointer, the signed pointer must be authenticated using the same key and context. If either the pointer (*sans* signature) or the context are different from the values used during signing, the authentication results in yet another invalid pointer. If the pointer, key, and context are the same values used during signing, the signature is stripped from the pointer, and the original (presumably valid) pointer value is restored. HAKC relies on this behavior to compute a context that was expected to sign a particular pointer, using a combination of information known at compile time and gathered during runtime.

Memory Tagging Extension: Memory tagging extension (MTE), introduced in ARMv8.5-A [4], allows for assigning a "color" or tag to a memory region, which can be used to segregate the address space into distinct regions. MTE introduces two instruction classes, one to assign a color to a memory region (shown in Figure 2), and one to retrieve the current color of a memory address. Given infinite tags, one could very simply create highly compartmentalized code — each compartment could be individually colored. However, MTE imposes some constraints on how it can be used: only 16 colors are available for use, the memory address must be aligned to 16 bytes, and the region to be colored can be no smaller than 16 bytes. The limited number of colors available makes simplistic compartmentalization inadequate, because the compartments are too broad in scope. The attack surface of only 16 compartments in the Linux kernel is large enough that bugs are unlikely to be mitigated. However, as this paper will show, the combination of PAC with MTE allows for the creation of significantly more compartments than the available colors. Colors are reused, but compartments are protected using PAC contexts computed from hard-coded values known at compile time and from the address tags retrieved during runtime in order to prevent reused colors from enabling spurious access.

B. Kernel Vulnerability Analysis

As inspiration for HAKC, we analyzed high severity (CVSS 3.0 rating 7.0 or higher) CVEs issued for the Linux kernel from Jan. 2015 through May 2021, and determined if the CVE could be mitigated with memory safety or compartmentalization. A CVE is considered mitigable if the presence of memory safety and/or compartmentalization would prevent the bug, and unmitigable if neither mechanism would prevent the bug. The basis for classification was determined by searching for keywords in the description that map to the two defense mechanisms (i.e., *arbitrary code execution* maps to compartmentalization, while *use-after-free* maps to memory safety), or manual analysis of patches in cases where the description was unclear. Figure 3 presents a summary of our findings. Out of the 567 CVEs in our dataset, 229 could be mitigated through compartmentalization, and 193 could be mitigated using memory safety. Only 71 could be mitigated by either defense mechanism, implying the continued importance of memory safety alongside compartmentalization, and only minimal overlap in protection when both are enabled.

There are 73 CVEs that are unmitigable with compartmentalization and memory safety, of which 57 involve incorrect or missing domain-specific logic, such as discarding returned error values or a cold path missing a data validity check. The remaining unhandled CVEs involve race conditions (9), integer over/under-flows (8), and a configuration that enables unsupported functionality (1).

III. Threat Model and Assumptions

In line with other kernel security mechanisms, we assume that an attacker does not have root access, and thus cannot modify kernel modules. However, they can take arbitrary actions in attempt to compromise a victim kernel module, including making arbitrary system calls or having peripherals send arbitrary data [82]. We also assume that the LKM itself is not malicious, but contains exploitable bugs. Kernel functionality outside of the victim LKM is part of the trusted source base, and we assume that data originating from the kernel is valid. Trusting data passed into the LKM could lead to a confused deputy attack, but preventing such an attack would require full code and data flow analysis in the kernel. Such an analysis is currently impractical, and thus we require the kernel to be a trusted agent (similar to the trusted core of a microkernel). Additionally, we include the core SoC in the trusted computing base, including its tagging and pointer authentication implementations, but IO devices are outside of our trusted components and can be malicious. Three exceptions to our hardware assumption, however, are direct memory access (DMA) actions, hardware glitching attacks [81], and side channel attacks, such as Spectre [38], Meltdown [51], or Rowhammer [35].

We do not assume any further virtualization or security layer that provides a level of trust, such as a hypervisor or verified microkernel. Instead, HAKC moves policy enforcement to the hardware, and removes the difficult problem of verifying trusted software [57]. HAKC is designed to run on bare metal, but is capable of running in a virtual machine provided an existing implementation of hardware features.

Listing 2 and Listing 3 is an example of two partial LKM implementations that conform to our threat model, but provide an arbitrary read and write. Listing 3 is dependent on Listing 2, and the programmer's intent is to only read from the defined arrays. However, due to a missing check on `idx`, if the user calls `ioctl` with `MSG_PUT` or `MSG_GET`, and an index outside the range of $[0, \texttt{SIZE}]$, then any address can be written or read (barring page permissions). The monolithic design of the kernel will simply allow these accesses, but HAKC prevents them by compartmentalizing the two LKMs.

```
100 static unsigned long *m1_counts;
    typedef struct msg {
102     long idx;
        unsigned long val;
104 } msg_t;

106 unsigned long m1_get(msg_t* m) {
        return m1_counts[m->idx];
108 }
    EXPORT_SYMBOL(m1_get);
110
    int m1_init(void) {
112     m1_counts = kmalloc(SIZE*sizeof(unsigned long));
    }
```

Listing 2: LKM 1 (Arbitrary Read)

```
200 static unsigned long counts[SIZE];
    extern unsigned long m1_get(msg_t*);
202
    static int m2_ioctl(struct inode *inode,
204                     struct file *file,
                        unsigned int ioctl_num,
206                     unsigned long ioctl_param) {
        msg_t *tmp;
208
        switch(ioctl_num) {
210         case MSG_PUT:
                tmp = (msg_t*)ioctl_param;
212             counts[tmp->idx] = tmp->val;
                break;
214         case MSG_GET:
                tmp = (msg_t*)ioctl_param;
216             tmp->val = m1_get(tmp);
                break;
218         default:
                return FAILURE;
220     }
        return SUCCESS;
222 }
```

Listing 3: LKM 2 (Arbitrary Write)

IV. HAKC Compartmentalization API and Enforcement

HAKC is built around two core contributions, which, when combined, are instrumental to its ability to establish isolation within the kernel without further virtualization: the *Compartmentalization Policy API* and a hardware-based *Compartment Enforcement Mechanism*. The Compartmentalization Policy API exposes primitives that allow the developer to establish a fine-grained compartmentalization policy on code and data, while the Compartment Enforcement Mechanism efficiently enforces the specific compartmentalization policy at runtime. We detail each here.

A. Compartmentalization Policy API

The HAKC Compartmentalization Policy API allows developers to assign code and global variables to compartments. Stack and heap variables are assigned to the same compartment as the code that allocates them. The compartmentalization policy also specifies allowed control flow between compartments. When control flow transitions between compartments, any required data is also automatically transferred.

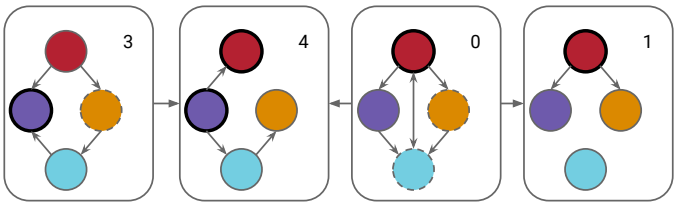
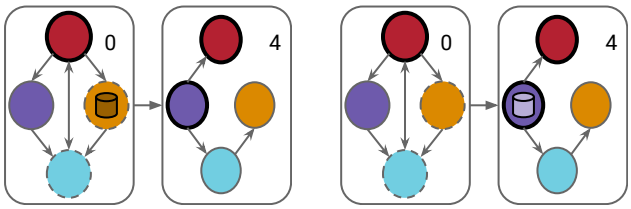


Fig. 4: An example compartmentalization involving four Compartments, each of which contains four Cliques. Edges between Compartments are allowable transitions. Bold Cliques are valid Compartment entry points, and dotted Cliques are valid Compartment exit points.



(a) Pre-Compartment Transition (b) Post-Compartment Transition

Fig. 5: An example Compartment transferring data ownership to an external Compartment. During this Compartment transition, orange data is recolored purple, and the data ownership is moved to the target Clique in the external Compartment. Upon return, the data is colored orange, and data ownership is restored.

The HAKC compartmentalization policy API allows users to configure the following in HAKC's novel two-level compartmentalization scheme:

1) *Cliques*, a partitioning of code and data into one or more groups, with each function and data object belonging to exactly one partition.
2) *Compartment*, a second level grouping of at least one Clique, with each Clique belonging to exactly one Compartment.
3) *Clique Access Policy*, the set of Cliques within a Compartment that a particular Clique can access, including itself.
4) *Compartment Transition Policy*, the set of Cliques that can legally transition control to outside the Compartment, and the set of allowable external Cliques that are valid control-flow targets.

Figure 5 shows two example Compartments. The red Clique in Compartment 0 can access the red, purple, blue, and orange Cliques, while in Compartment 4, red can only access itself. Figure 4 illustrates an example of several Compartments, and the allowable transitions; Compartment 0 can transfer control flow to Compartments 1 and 4, but Compartment 3 can only transfer to Compartment 4.

The two-level compartmentalization policy API introduced by HAKC provides three key benefits: 1) the ability to create a large number of compartments with a limited number of colors — overcoming the classic limitation of few tag bits; 2) efficient access to data defined in a Compartment, i.e., local data optimization; and 3) the flexibility for the developer to make fine-grained security/performance trade-offs. The first benefit frees the developer from practical limitations present in any commodity tagged hardware when designing compartmentalization policies. The number of desired compartments will exceed the number that any hardware-only system can supply (e.g., 2^{tag_bits}), inspiring our new two-level scheme that allows tag bit reuse across Compartments. As we will show later, grouping Cliques into a Compartment allows for creating several orders of magnitude more compartments than available tags would otherwise allow.

The second benefit allows for easier policy creation, and more efficient policy validation checks. While a Clique is executing, any data that is accessed by a pointer must satisfy two conditions: 1) the data must belong to the Compartment in which the Clique resides; and 2) the data must belong to a Clique the current Clique is allowed to access under the Clique access policy. These conditions are checked at runtime prior to the first dereference of a pointer in a function, but are not checked again unless the pointer is modified. Satisfying these two conditions ensures that arbitrary data access is prevented, and that data ownership is enforced. These conditions also enable faster checks as only the Clique access policy needs to be checked, and that policy only concerns 2^{tag_bits} Cliques, allowing a highly optimized implementation compared to the Compartment access policy.

To illustrate the third benefit — fine-grained security/performance trade-offs — we first describe how Clique and Compartment access policies work. The developer establishes a Compartment by partitioning code and data into one or more Cliques, determining which Cliques each particular Clique should legally access, and which Compartments are valid control-flow targets. The specific compartmentalization policy can be determined manually, or automatically through static or dynamic analysis, and can be different for different Compartments. All data and code in a Compartment must belong to exactly one Clique, and directed edges between Cliques indicates valid access. The directed edges represent a forward-edge Clique-based control-transfer policy, and does not need to be symmetric. For instance, the developer might want a green Clique to call a function in a red Clique, but forbid the red Clique from calling a green Clique.

When control flow must exit a Compartment, through either a direct or indirect function call, then the ownership of data that exits must be transferred to the target destination, and then restored upon return. The transfer ensures that data checks in Cliques can proceed as intended, which maintains valid data ownership. In the case of indirect function calls, the target is checked to ensure that it conforms to the valid transition policy that the Compartment defines, and that the target is a valid entry to the target Compartment. If the target of an indirect call is within the same Compartment, the access policy for the current Clique must be followed, but no data ownership is transferred. The control-flow checks ensure a valid control path is followed, and arbitrary code execution is prevented.

By adjusting how code and data are partitioned into Cliques and Compartments, the developer can make fine-grained trade-offs between security and performance. As the number of Compartments increases, i.e., the more compartmentalized the kernel becomes, the harder an attack becomes, because the attacker has to find a valid control-flow path that obeys both the Clique access policy and Compartment control-transfer policy. However, the increased Compartment count necessarily leads to more data ownership transfers, which can incur large performance overhead. HAKC allows developers to specify fine-grained boundaries to suit their particular performance and security needs.

B. Compartmentalization Enforcement Mechanism

While Clique code is executing, HAKC does not rely on any additional TCB, but instead uses hardware for access policy enforcement. Prior compartmentalization mechanisms

rely on additional layers of abstraction, e.g., kernel code for user-space compartments [31], [78], [93], or hypervisors for kernel code [55], [56], [60], [73]. Breaking this "turtles all the way down" paradigm for compartmentalization by rooting trust in hardware avoids adding layers of abstraction and growing the TCB. HAKC is the first to solve this challenge for realistic, commodity hardware.

In order to provide compartmentalization, HAKC needs to be able to partition the virtual address space separately from traditional paging, and the ability to associate a pointer with metadata of bit-size larger than the available address space partitions, referred to as the pointer's *conjoined metadata* (CM). By virtual address space partitioning, we mean a method of designating a virtual memory address range as distinct from the rest of the address space. Tagged architectures, which typically provide a small number of bits to associate with (or *color*) a virtual memory address range, are an existing partitioning method. All pointers, either statically created at load time or allocated dynamically, are associated with a specific CM encoding which Clique owns the underlying data. The association must be hard to compute given the pointer and CM. HAKC enforces partition access policies by, for every pointer accessed during runtime, computing a *candidate CM* using the address space partition information (which equates to Clique membership), and Compartment information. Before the pointer is accessed, the candidate CM is compared with the pointer's actual CM. If the runtime information causes the candidate CM to differ from the actual CM, then the pointer dereference cannot happen.

As long as the hardware provides address space partitioning and pointer CM association primitives, then HAKC guarantees that the defined compartmentalization policy is followed. If some bug inside the compartment modifies a pointer such that it points to data that violates the access policies defined for either the Clique or Compartment, the candidate CM will differ from the pointer's CM. Accordingly, if some bug outside the Compartment modifies a pointer that is currently being used by the Clique to violate access, the candidate CM will again differ from the correct CM. In both cases, data access is prevented, and compartmentalized code is prevented from accessing data not explicitly granted to it. While we implemented HAKC using ARMv8.5-a, HAKC is not tied to any specific architecture. Any mechanism that provides the necessary partitioning and association primitives may implement HAKC.

C. Example Case Study

Here we describe how HAKC can compartmentalize the two LKMs listed in Listing 2 and Listing 3. In this example, all code and data in Listing 2 will be in the same Black Clique in Compartment 1 (referred to as $(1, Black)$), while the code and data in Listing 3 will be in the Gold Clique in Compartment 2 ($(2, Gold)$).

When `m2_ioctl` executes, it first validates that `tmp` is accessible by checking its $(2, Gold)$ CM with the candidate CM computed with runtime data. `m2_ioctl` will also check if the $(2, Gold)$ CM for `counts + tmp->idx` matches the computed candidate CM, and, if `m1_get` is called, will recolor `tmp` to black, and associate its value with $(1, Black)$. When `m1_init` executes, it associates `m1_counts` with $(1, Black)$. Finally, when `m1_get` executes, it checks `m` and `m1_counts + m->idx` with $(1, Black)$.

If an attacker directs control flow to `m2_ioctl`, then `ioctl_param` must be properly associated with $(2, Gold)$, which is computationally hard to perform. Similar conditions must be satisfied for `m1_get`. Additionally, if a bug outside of $(1, Black)$ changes the value of `m1_get` to point outside the Compartment, then the pointer will not be associated with $(1, Black)$, and HAKC prevents its dereference.

V. Compartment Policy and Enforcement Mechanism Implementation

Here, we detail how HAKC enforces the data and control-flow policies that provide isolation to compartmentalized code. The HAKC Compartment Enforcement Mechanism uses a combination of tagged architecture and cryptographic hashes to provide access enforcement. Namely, the Compartment Enforcement Mechanism uses Arm's MTE to provide tagging support, and PAC to provide cryptographic hashing. See § II for details. PAC is used to ensure that pointers have not been tampered with inadvertently, and that they conform to the various access-control policies defined for Cliques and Compartments, while MTE provides the runtime Clique membership. By combining information known at compile time, such as a Compartment identifier and access-control policies, with MTE colors gathered dynamically, HAKC can provide significantly more compartmentalization granularity than the 16 compartments natively provided. HAKC recycles colors in different Compartments, but the compile time information effectively creates "hues" of the available colors to allow for a large number of compartments.

Cliques: A Clique belongs to exactly one Compartment, and combines code and global, stack, and dynamically allocated data into a logical group, all of which is assigned a color, C_c. C_c needs to be unique to the Compartment the Clique belongs to, but it does not need to be globally unique. In fact, one of the primary contributions of this paper is a design that allows for the safe multiplex use of colors in different compartments. A good example of the type of information that a Clique contains is what is defined in a typical C source file: exported and static functions and global variables, stack allocated objects, and dynamically allocated memory. We do not, however, force all functions or data in a source file to belong to the same Clique, and the developer is free to partition as they see fit.

Using their specific C_c, Cliques also define two tokens, Tok_{acl} and Tok_s, used in authenticating a pointer and signing pointers respectively. Tok_{acl} is used to generate the context used for PAC authentication, and encodes both the Clique's Compartment identifier, ID_n, and allowable Clique code and data accesses. Tok_s encodes ID_n and C_c, and provides the PAC context when signing pointers. We will detail how these tokens are used to enforce compartments in § V-A.

Compartments: A Compartment consists of at least one Clique but no more Cliques than the number of available tags, N_{tag}, and is assigned a globally unique identifier, ID_n. All data accessed by any Clique must belong to the Compartment, and the identifier is used to ensure that is the case during pointer authentication.

TABLE I: Clique and Compartment properties.

Clique	Compartment
Access Token (Tok_{acl})	Entry Token (Tok_{ent})
Signing Token (Tok_s)	Valid Compartment Targets (T_n^*)
Color (C_c)	Unique Identifier (ID_n)

Additionally, a Compartment defines an entry token, Tok_{ent}, which encodes the Cliques that can be targets of indirect jumps, along with ID_n. Tok_{ent} is known to all other Compartments that could potentially execute code in a Clique. Similarly, a Compartment must know all valid potential Compartments to which it could transfer control flow. Therefore, a Compartment maintains a mapping of valid Compartment ID_n and the respective Tok_{ent} in T_n^*. Before an indirect call is executed, the target function is checked that it belongs to a valid Compartment, and is a valid entry Clique using each entry token in T_n^*. Figure 4 is an example of several Compartments, along with their allowable Compartment transitions. In this instance, control flow is able to transfer from Compartment 0 to either Compartment 1 or 4, but not Compartment 3. Additionally, if control flow is going from Compartment 0 to Compartment 4, the target Clique *must* be orange or red, and cannot be green or purple. Cliques in different Compartments have different access-control policies, yet share colors.

Data Access Policy: All data accessed by a Clique through a pointer must belong to the currently executing Compartment, and be validly accessible according to the current Clique access-control policy. PAC is used to validate both conditions hold. Pointers are signed using Tok_s, and when authenticated later, Tok_s is calculated using Tok_{acl}, and the pointer target MTE color. If the pointer is erroneously manipulated, or points to data that is either the incorrect color or does not belong to the Compartment, then the computation of the PAC authentication context will differ from Tok_s and will thus fail authentication.

Compartment Transitions: When a Compartment needs to transfer control to another Compartment, two actions need to occur: 1) the target must be validated against T_n^*; and 2) data ownership must be transferred to the target destination. In the event of a direct function call to outside the Compartment, only step 2 needs to occur.

Function pointers are signed by their Clique using their Tok_s, and, similar to how data pointers are authenticated, that same token is computed using the target MTE color and the Tok_{ent} from T_n^*. Each Tok_{ent} in T_n^* is used to compute a possible PAC authentication context. If the authentication of the signed function pointer succeeds with the computed context, then control flow is allowed to the target Clique in the different Compartment. If authentication fails, then a different Tok_{ent} is tried until all valid transitions are exhausted, at which point control flow to the target is prevented.

If the Compartment transition is allowed, all pointer arguments (and any submember pointers) are recolored the target Clique color, and the pointers are resigned using the target Clique Tok_s. The act of recoloring and resigning pointers transfers ownership of data from the current Clique to the target Clique, and pointer validity happens within the new Compartment as previously described. When the indirect call returns, data ownership is restored to their previous state prior to the indirect call.

TABLE II: Computed PAC context used for access enforcement for a pointer `p` used by Clique M.

Operation	PAC Context
Transfer to M	$Tok_{s,M}$
Data Access Check	$Tok_{acl,M} \wedge V(C_p)$
Valid Transfer to Compartment G	$Tok_{ent,G} \wedge V(C_p)$

We note that only transferred pointers are resigned (and retagged). No effort was made to automatically invalidate any aliases of transferred pointers. This means that lingering aliased pointers in the origin Clique could allow access to data owned by another Clique of the same color, because the original signature would be valid. Pointer struct members and pointers to objects allocated on the current stack frame are resigned and protected, but HAKC does not solve the general aliasing problem. We rely on the programmer to invalidate aliased pointers when necessary.

A. Access Enforcement

For both signing and authentication, PAC takes as input a pointer and a user-specified 64-bit context. For a signed pointer to pass authentication, the *exact* context used to sign the pointer must be provided. In order for HAKC to ensure that the signing context can be correctly provided when authenticating, all tokens (e.g., Tok_{acl} or Tok_{ent}) have the same general form:

$$Tok_{i,n} = ID_n \oplus (V(C_i) \vee \cdots) \tag{1}$$

where $V(C_i)$ is a bitvector of size N_{tag} with the bit corresponding to color C_i set to 1 and all other bits set to 0. In other words, HAKC tokens consist of the particular Compartment identifier concatenated with one or more vectorized color values bitwise `OR`'d together. The composition of colors with Compartment identifiers is what allows for the reuse of colors in different Compartments, and the creation of far more compartments than N_{tag}, while still providing strong compartmentalization guarantees. The number of compartments potentially available is $2^{64-N_{tag}} \cdot N_{tag}$, which for MTE ($N_{tag} = 16$) equates to $4 \cdot 10^{15}$ compartments. Dynamically allocated data is immediately transferred to the Clique which created it, and stack-allocated data passed to compartmentalized functions must be transferred to the current Clique if not proven safe (see § V-D). All other data must have been previously signed, either by the kernel when control first flows into the Compartment, or by other Cliques during a cross Compartment transfer.

Signing tokens, which always belong to a specific Clique, utilize only one vectorized color: their own. Access tokens (i.e., Tok_{acl} or Tok_{ent}) are comprised of the relevant allowable colors. For example, in Figure 5, the purple Tok_{acl} in Compartment 0 consists only of the vectorized blue and purple values `OR`'d together, while the Compartment 0 red Tok_{acl} has all four color values set to 1. Similarly, in Figure 4, Tok_{ent} for Compartment 4 will be composed of the vectorized purple and red, as those are the valid entry Cliques to that Compartment (indicated with bold outlines).

TABLE III: Summary of needed developer effort and automated instrumentation provided by the LLVM pass.

Operation	Developer	LLVM
Compartmentalization policy definition	✓	
Data ownership transfers from kernel to Compartment	✓	
Dynamic memory allocation transfers	✓	
Data access validity check insertions		✓
Valid Compartment transition check insertions		✓
Data ownership transfers to external Compartment		✓
Data ownership transfers from external Compartment		✓
Signature stripping in unprotected code		✓
Ensure all dynamic allocations are transferred		✓
Signing of global variable addresses		✓

To confirm that a Clique can access data at a signed pointer, a candidate signing token is formed by computing the color vector of the underlying pointer data, and concatenating ID_n. The candidate signing token is then bitwise `AND`'d with Tok_{acl} to provide the PAC authentication context. If the Clique is allowed access to the pointer color, and the pointer was signed by the current Compartment, then the bitwise operation results in the exact context used to sign the pointer, and PAC authentication succeeds. If data belongs to a different Compartment, but is colored an accessible color, PAC authentication fails, because the upper bits of the computed PAC context are different from the signing context. Likewise, if data belongs to the Compartment, but colored an inaccessible color, PAC authentication also fails, because the lower bits of the computed PAC context differ from the signing context. Only *valid* control and data flows are allowable inside the Clique, and that is enforced through requiring valid signatures before dereference. Table II presents a summary of compartmentalization operations, and the PAC context computations.

B. Developer Effort

Manual development is limited to specifying a compartmentalization policy, transferring dynamically allocated data to a Clique, and specifying which kernel data needs to be transferred to a Clique entry function before its invocation. While the annotations are lightweight (the largest single annotation we made is 74 lines to transfer data into a Clique), we have it as future work to automate these efforts.

The pointer validity checks and Compartment transitions are added via an LLVM [45] pass, which runs on annotated sources and skipped otherwise. Compiling the Linux kernel using LLVM is supported, and no custom modifications to the LLVM source were needed. To establish a Clique, the LLVM pass places all similarly colored code and data into specially defined ELF sections, which the kernel module loader looks for when loading. When the compartmentalized LKM is first loaded, the kernel colors the code and data appropriately before any initialization code is executed. Once initialization code begins executing, the kernel performs its typical page-level permission enforcement in addition to the compartmentalization enforcement provided by HAKC. However, the page enforcement prevents changing the colors of code and read-only data, and for safety HAKC does not enable write permissions when changing colors. Therefore, the developer must be aware of these limitations when developing the access-control policy for Cliques.

As mentioned earlier, HAKC computes a PAC signing context using some compile-time information, specifically in which Clique and Compartment particular data and code belongs. Currently, the developer must annotate code and global data in the source to establish the Clique and Compartment membership. The annotations make the compartment membership permanent, since, for example, Compartment identifiers become encoded in instructions moving immediate values into registers. However, this is purely a performance optimization to avoid additional memory lookups, and HAKC can be extended to dynamically change compartmentalization policies at runtime.

In the simplest case where all functions and data defined in a source file belong to the same Compartment and Clique, the annotations are very lightweight; only a single addition of a macro defining the Compartment and Clique at needs to be added. In a planned future iteration, the LLVM pass will perform this operation for the user. If further partitions are required, the developer simply annotates the partitioned code or data, while the rest remains in the original partition. HAKC makes no attempt to optimize the developer-established partitions, The performance of the final system could be highly dependent upon the chosen partitioning, since Compartment transitions can be expensive due to the recoloring and pointer resigning process involved (see § IV). We leave it for future work to determine effective partitioning strategies, but HAKC supports any partitioning as long as the Cliques count in a Compartment does not exceed N_{tag}.

Finally, kernel module code is executed using function pointers registered to the kernel by the module during initialization. These function pointers represent the functionality the LKM implements. For example, a filesystem LKM implements a filesystem-specific `read` function, and registers that function with the kernel, which the kernel then invokes when reading the filesystem is required. Since all functions in a Clique expect all dereferenced pointers to be properly signed and colored, before executing compartmentalized code, the kernel must transfer the data to the target Clique. The developer must write a function to transfer input data, and provide that function to the kernel instead of the original function. Table III provides a summary of all developer efforts and LLVM pass actions.

C. Policy Creation

The modules that we compartmentalized in our proof-of-concept implementation imposed a very simple compartmentalization policy. Namely no more than two Cliques belong to a Compartment, and control flow outside a Compartment was limited only to the kernel. While such a policy is easy to define, and provides some protection, users will likely desire a more complex and automatically generated compartmentalization scheme. As mentioned in § V-B, a more compartmentalized kernel makes attacks harder, but could affect performance due to more Compartment transitions that need to occur. Unfortunately, developing a performance optimal compartmentalization policy is a similar to the Partition Problem, which is NP-Complete [19]. Therefore, we do not expect a general solution to optimal compartmentalization to be found. However, like solutions to the Partition Problem, we expect heuristics and dynamic analysis can provide good enough, if suboptimal, solutions for real world applications. Little work has been done on bare-metal commodity kernel compartmentalization, and we

leave it for future work to develop strategies for automatic policy generation.

D. Optimizations

HAKC utilizes two major optimizations to achieve its low overhead, an interprocedural analysis that seeks to prove pointers have been validated by all caller functions, and an intraprocedural analysis for efficient data check placements. We detail each optimization here.

Interprocedural Optimization: Many functions in the Linux kernel are valid only for a single compilation unit, i.e., functions defined using the `static` keyword. Because no other code outside of its source file can rely on these static functions, for any such function F, we can determine the set of pointers dereferenced in F that all caller functions authenticate prior to calling F. Those pointers then do not need to be authenticated, and caller functions can provide the authenticated pointer values instead of the signed versions. To ensure that our analysis can determine the full set of dereferenced pointers, we schedule our LLVM pass late in the compilation process after function inlining. The analysis HAKC performs here is similar to live variable analysis, with every function maintaining a set of pointers, $P_{start,F}$, that are known to be authenticated at F's entry, as well as the set of pointers F authenticates, $P_{auth,F}$. P_{start} is initially the empty set for all functions, and let $P_F = P_{start,F} \cup P_{auth,F}$. At every call site to F, we check if each pointer argument, p, is in the caller function's P, and if the pointer argument is in all P, then $P_{start,F} = P_{start,F} \cup p$. This analysis repeats until a steady state is achieved, and no P_{start} changes. For the LKMs we compartmentalized, this analysis reduced the number of data check insertions by 2%. The number of global and stack variables that needed to be signed, because their addresses are passed to other compartmentalization functions, is reduced by 8%. These reductions translate to 12% fewer data authentication checks, and 19% fewer transfers when performing the 100KB overhead experiment detailed in § VI-B.

Intraprocedural Analysis: A naïve approach to authenticating pointers would involve finding the set of dereferenced pointers, and placing the authentication of each such pointer in the first basic block of the function. However, some pointers are only dereferenced when specific conditions are satisfied, and thus authentication of those pointers needs to happen only when they will actually be dereferenced. Therefore, we place each pointer authentication only at the immediate dominator basic block of all said pointer uses. We also ensure that if a pointer use is within the same basic block as the authentication, then the authentication happens before the dereference. We only create one authentication per dereferenced pointer in a function, which can lead to some unnecessary overhead. If a pointer gets dereferenced in two basic blocks whose immediate dominator is different from either, then the authentication can happen without the pointer dereference taking place. This can occur, for example, when pointers are only dereferenced in error handling with a `goto` statement for final cleanup.

VI. Evaluation

When evaluating HAKC, we wanted to answer the following research questions:

```
ldg xT, xN
  ldr x16, [xN]
  mov x17, #0xF0
  lsl x17, x17, #49
  and xT, x17, x17
```

```
stg xT, xN, imm
  ldr x16, =TAG_MEM
  mov x17, xT
  lsr x17, x17, #49
  str xT, [x16]
```

Fig. 6: MTE Instruction Analogs

1) What is the overhead imposed by HAKC?
2) What is the overhead of using multiple Compartments in a single system?
3) Will users notice any difference in performance under real-world work loads?

Here we describe our experimental setup, as well as address our research questions through microbenchmark and simulated user browsing experiments. We performed all evaluations on a Raspberry Pi 4 8GB, and our kernel version was based off the Debian 5.10.24 source.

A. Instruction Analogs

As of June 2021, no hardware implementing MTE is available, and the most readily available hardware implementing PAC are Apple devices containing the A12 processor, and are unfortunately heavily locked down. Therefore, in line with the evaluation methodology of Liljestrand et al. [50], we ensure correct functionality using emulation, and measure overhead using instruction analogs in lieu of PAC and MTE instructions. An instruction analog is a series of instructions that consumes the same CPU cycles and the same memory footprint as the PAC/MTE instructions, but does not perform an actual check. Thus it can be used for accurate performance evaluation without a PAC/MTE-enabled processor. The PAC analogs are adapted from the PARTS system detailed in [50], and we detail the MTE analogs here.

Version 5.10 of the Linux kernel uses the load and store instructions for single or multiple tags, namely `ldg`, `stg`, `ldgm`, and `stgm` respectively. For the single tag instructions, the kernel only uses the post-index encoding. To simulate the `ldg` instruction, which writes the tag to bits 49–53 of an input register, we perform a load of the target address, and finally place a valid tag value in the appropriate register bits. To store a tag, we retrieve the tag from bits 49–53 of the pointer address, and write to a global variable. Multiple tag operations simply repeat these single tag operations. We took care to ensure that memory accesses occur at every MTE instruction to simulate a worse case memory tag access, but an actual implementation of MTE could include tag caching or other performance enhancements. Thus, we claim that our performance overhead is an estimation of *worst case* performance. The instruction substitutions are listed in Figure 6.

B. Single Compartment Performance Overhead

To measure the compartmentalization overhead, we compartmentalized the `ipv6.ko` LKM into a single Compartment with two Cliques. Both Cliques were given access to each other's code and data. We used ApacheBench to retrieve a 100KB, 1MB, and 10MB file 1000 times from an unmodified

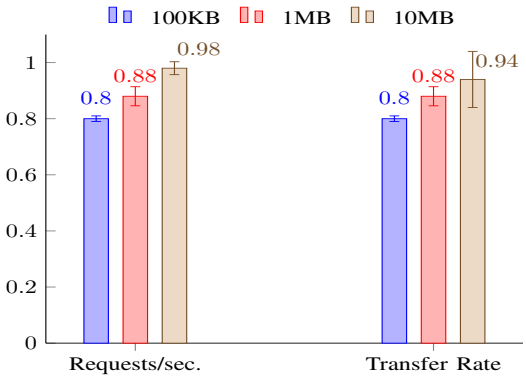


Fig. 7: `ipv6.ko` overhead normalized to unmodified kernel when transferring various sized payloads.

Apache server running on the Raspberry Pi. We repeated each experiment 10 times, and recorded the reported requests/sec and transfer rate in MB/s. We measured all performance overhead relative to the unmodified kernel, and both kernels sharing the same user-space.

The overhead measurements for `ipv6.ko` are listed in Figure 7, normalized to the performance of the unmodified kernel. Overall, the performance of our `ipv6.ko` compartmentalized LKM is good compared to the baseline, with only a 20% reduction in both requests per second and transfer rate in the worst case.

When the transfer size is small, the establishment of the TCP connection imposes significant overhead relative to the actual transferring process. Once the TCP connection has been established, however, relatively few data checks need to be performed to transfer the payload. This explains the low 2%–4% overhead for the 10MB payload measurement; larger payloads spend less time establishing the TCP connection relative to the total transfer time. Figure 8a shows this behavior in the HAKC operations per kilobyte Apache sends. HAKC operations include the number of Compartment transitions, the number of data pointer authentications, and the number of code pointer authentications. While the number of operations per second either increase or remain constant, the number of operations per KB of transmitted data monotonically decreases with payload size.

C. Multiple Compartment System Overhead

`nf_tables.ko` implements a packet filtering mechanism within the Linux kernel. We compartmentalized this LKM by placing all code and data in a single Clique using a different color and Compartment from those used in the `ipv6.ko` LKM. The `ipv6.ko` and `nf_tables.ko` LKMs were not allowed to transition to each other directly. Since there is only a single Clique, no further compartmentalization policy needed to be specified.

To measure the overhead of using both LKMs on the same system, we defined a packet filter rule that drops packets with a source address from a specific IPv6 address. We then ran our microbenchmark detailed in § VI-B using the unmodified kernel, the compartmentalized kernel with only the `ipv6.ko` LKM compartmentalized, and the compartmentalized kernel with both HAKC LKMs enabled. The results, normalized to the unmodified kernel (U) and `ipv6.ko`-only (S) kernel overheads, are listed in Figure 9.

TABLE IV: The measured time differences between the compartmentalized kernel of the lowest and highest standard deviations of unmodified kernel load times. Negative delta numbers indicate slower compartmentalized load time.

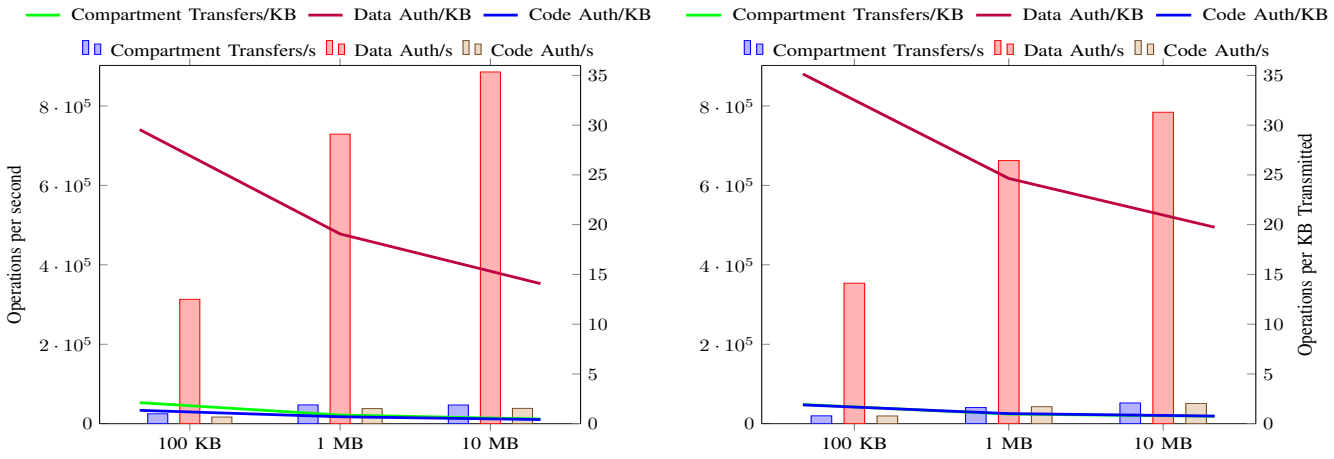
Website	Delta (s)	Stdev (s)
linkedin.com	-0.47	0.065
hdfcbank.com	-0.12	0.085
google.cn	-0.068	0.086
bing.com	-0.087	0.13
investing.com	38	62
okezone.com	-11	20
cnn.com	-9.8	15
yahoo.com	-4.9	15

The general trend regarding payload size and overhead shown in Figure 7 is again present for the overhead against the unmodified kernel. However, the performance relative to the single Compartment system degrades with payload size. The performance degradation comes from the additional Compartment transitions the kernel makes to perform both packet filtering and TCP functionality with every TCP ACK packet received. This behavior is shown in Figure 8b, with the higher number of data pointer authentications per kilobyte than with just IPv6 compartmentalized. Regardless, Figure 9 shows a linear growth of 14%–19% per compartment *when the compartments are related, but provide orthogonal functionality*. Compartmentalizing both the IPv6 and packet filtering represents a worst case for performance loss, since all HAKC operations for both LKMs will occur in tandem, and will thus be directly compounded. A better compartmentalization policy will likely amortize individual overheads to a lower total overhead, but we leave that evaluation for future work.

D. User Website Browsing

Using ApacheBench to measure raw performance does not necessarily provide a good indication of whether a user will notice any performance difference when using the compartmentalized kernel for everyday activities. For example, activities unrelated to the kernel networking stack, such as routing delays, website rendering, or advertisement negotiation, can add significant time to end-user web page loading. To answer Research Question 3, we want to measure any significant difference in IPv6 website loading time between using the unmodified kernel and our compartmentalized `ipv6.ko` LKM given these external factors.

To that end, we created a Selenium script that spawns a headless Firefox instance, and proceeds to play a specific YouTube video, and then visits the 50 most popular websites (as determined by the Alexa Top 1M) that advertise an IPv6 address in their DNS Authoritative Record (an `AAAA` entry). We disable all memory and disk cache use and enable IPv6 use in Firefox. Additionally, before retrieving each website, we delete all cookies, and perform a DNS query to ensure that ISP DNS entries are fresh. Afterwards, we measure the time the Selenium web driver takes to fully render the page, or the time the YouTube video takes to complete. To account for possible differences in advertisements, we retrieve each website using



(a) Only IPv6

(b) IPv6 and `nf_tables`

Fig. 8: Average HAKC operations per second and per KB transmitted while running ApacheBench.

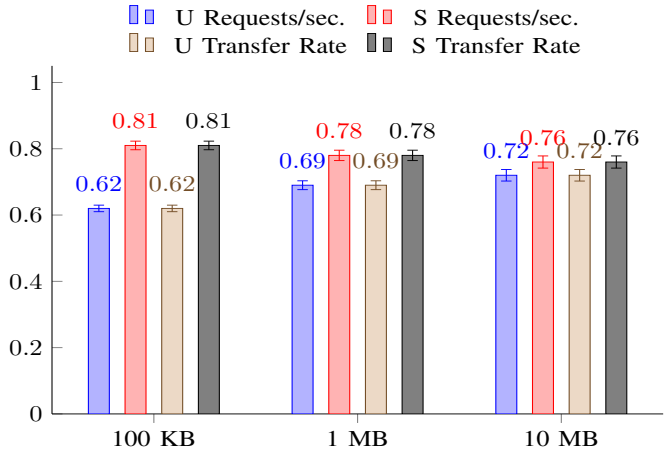


Fig. 9: Overhead imposed when using multiple Compartments in a single system, normalized to the unmodified kernel (U) and single Compartment systems (S).

```
int ip6_find_1stfragopt(struct sk_buff *skb, u8 **
    nexthdr)
{
    u16 offset = sizeof(struct ipv6hdr);
    struct ipv6_opt_hdr *exthdr =
          (struct ipv6_opt_hdr *)(ipv6_hdr(skb) + 1);
    unsigned int packet_len = skb_tail_pointer(skb) -
                              skb_network_header(skb);
    /* ... */
    while (offset + 1 <= packet_len) {
        struct ipv6_opt_hdr *exthdr;
        switch (**nexthdr) {
            /* ... */
        }
        offset += ipv6_optlen(exthdr);
        *nexthdr = &exthdr->nexthdr;
        exthdr = (struct ipv6_opt_hdr *)
             (skb_network_header(skb) + offset);
    }

    return offset;
}
```

Listing 4: CVE-2017-9074

the unmodified kernel and compartmentalized kernel in turn before retrieving the next website. We repeated this experiment 5 times, with each retrieval separated by approximately 1 hour.

Overall, we measured the average load time of the compartmentalized kernel to be 1.19 ± 4.34 seconds slower than the unmodified kernel. Because the standard deviation of load time differences is much larger than the average, we conclude that the compartmentalized kernel is not significantly different from the unmodified kernel, and that a user will not notice a difference using a compartmentalized kernel.

Despite our efforts to mitigate any possible difference between website retrievals, we did measure large differences in load times of some websites, on both large and short time retrieval spans. For example, `investing.com` would sometimes load in 4 seconds, and then after rebooting into a new kernel, the website would take 151 seconds. For this reason, we did not include `investing.com` in the average cited above. We were unable to determine any correlation between time of day or kernel type; the same website would be slow for the unmodified kernel at one time, and similarly slow for the compartmentalized kernel at another time, while the different kernels would statistically tie at every other time. The websites that exhibited the highest variance are those that serve dynamic content, such as `cnn.com`, while the lowest variance websites, such as `hdfcbank.com`, do not. We attribute the high variance to the underlying dynamic content generation on the server side, i.e., outside of our control.

Table IV lists the websites with the smallest and largest unmodified kernel load time standard deviations, along with the measured time differences when using HAKC. In total, 20% (10/49) of the websites were measured to be faster using the compartmentalized kernel, and in all but one case, the load time delta was within 2 standard deviations (95% confidence). This provides further evidence that HAKC compartmentalization would go unnoticed by users in everyday usage.

```
void mwifiex_set_uap_rates(
     struct mwifiex_uap_bss_param *bss_cfg,
     struct cfg80211_ap_settings *params) {
    struct ieee_types_header *rate_ie;
    /* ... */

    rate_ie = (void *)cfg80211_find_ie(
     WLAN_EID_SUPP_RATES, var_pos, len);
    if (rate_ie) {
        memcpy(bss_cfg->rates, rate_ie + 1,
               rate_ie->len);
        rate_len = rate_ie->len;
    }

    rate_ie = (void *)cfg80211_find_ie(
            WLAN_EID_EXT_SUPP_RATES,
            params->beacon.tail,
            params->beacon.tail_len);
    if (rate_ie)
        memcpy(bss_cfg->rates + rate_len, rate_ie + 1,
               rate_ie->len);

    return;
}
```

Listing 5: CVE-2019-14815

E. Security Evaluation – CVE Case Studies

Here, we will provide a security evaluation on two real-world bugs, chosen to illustrate HAKC protection against bugs within and outside of compartmentalized code: CVE-2017-9074 [66] and CVE-2019-14815 [67]. CVE-2017-9074 is an internal IPv6 bug, while CVE-2019-14815 is an external bug in the Marvell Wifi driver. Of the 567 CVEs in our analysis set (see § II-B), only 12 involved IPv6, demonstrating the importance of having compartments be hardened against external bugs, as most kernel bugs will be outside of a compartment.

CVE-2017-9074 (Listing 4) allows for reading memory outside the bounds of the intended object. The bug involves a missing check on `offset` against `packet_len` that ensures that the code is reading within the bounds of the socket buffer, `skb`. Through a series of system calls, a malicious user can craft an IPv6 packet that contains an invalid option, which causes `offset` to be much larger than the size of the allocated buffer for `skb`. `offset` is used to compute `*nexthdr`, which is read in the switch statement. This read is the out-of-bounds memory read.

HAKC prevents arbitrary out-of-bounds memory accesses like this, and instead limits the code's ability to only access the data explicitly allowable by the Clique `ip6_find_1stfragopt` belongs to. The large, corrupted `offset` value can place `exthdr` in one of several places: 1) a different Compartment and a different colored Clique; 2) a different Compartment but the same colored Clique; 3) the same Compartment and a different colored Clique; and 4) the same Compartment and the same Clique. In the first two situations, PAC authentication will fail because the computed PAC context will not match the PAC context used to sign `exthdr`. The third situation allows access only if the Clique is accessible according to the defined access-control policy, and the fourth situation will be allowed by HAKC.

To successfully perform this out-of-bounds read on HAKC-protected code, the attacker would have to construct `offset` such that the resultant pointer points to an accessible Clique, *and* contains the correct signature. The first condition already limits arbitrary accesses, and the second condition is computationally hard. This is how HAKC compartmentalizes code and data. The attacker is able to only access data allowed by the access-control policy, even in the presence of bugs, *and* the attacker must perform a computationally hard task to do so.

CVE-2019-14815 (Listing 5) is a bug in the Marvell Wifi driver that uses data from user-space in `memcpy` without checking the data length, leading to a heap overflow. Assume that the attacker uses this CVE from uncompartmentalized code to overwrite a pointer in compartmentalized code. The new pointer must again conform to all data access policies, and must contain a valid signature for the new pointer. Only if the new pointer is validly accessible and correctly signed, then the attack will succeed. However, as mentioned earlier, satisfying all the conditions is computationally hard.

Unfortunately, non-pointer compartmentalized data can be corrupted. However, this will likely only cause a denial of service, which, though severe, is considered less serious than privilege escalation. One mitigation would be to utilize the "traditional" MTE, and store the color in the pointer along with the PAC signature. The MTE hardware can retrieve the color of accessed addresses, and check that value with the stored value, and throw a fault if they mismatch. The use of MTE and PAC in this way reduces the available signature bits by half, making brute force guessing of a signature easier.

VII. Discussion and Threats to Validity

Here we discuss Hardware-Assisted Kernel Compartmentalization security and performance limitations.

A. Security Limitations

HAKC does not prevent all attacks. An attacker might find a valid control-flow path that adheres to all Clique and Compartment access policies, yet allows the corruption of data within a Clique. However, that corrupted data pointer cannot belong to some invalid Clique when dereferenced, and thus the damage the bug causes is contained to the compartmentalized code. Additionally, data provided by the kernel is assumed to be valid, which can lead to a confused deputy exploitation. Some bug in the kernel can allow invalid data to be signed and passed to compartmentalized code. Unfortunately, no practical solution to this problem, beyond formal verification [37], has been found. Instead, we envision a potential solution: formally verify the memory management and IPC code [41], and make all other functionalities HAKC-protected LKMs. Such a system could provide microkernel-like security, while keeping the robust functionality of existing kernels.

B. Performance Limitations

As indicated in § VI-C, LKMs that compute on the same data compound their overheads in the worse case. We have it as future work to evaluate the performance overhead of compartmentalizations that are largely unrelated. For example, we hypothesize that the overhead introduced by compartmentalizing `ipv6.ko` will not affect a compartmentalized Bluetooth LKM, and the overall system overhead will be the maximum overhead of either compartmentalized LKM. We have it as future work to evaluate more compartmentalizations, and their effect on overall system overhead.

During the development of HAKC, we theorized strategies to reduce overhead of compartmentalization. For example, minimizing recolor operations by coloring all entry Cliques the same color might reduce overhead, since pointers only need to be resigned with the target ID_n. Additionally, static or dynamic analysis might indicate efficient compartmentalization policies. Existing tools, such as the Syzkaller [29] fuzzing engine or the KUnit unit testing framework, can provide insight into novel compartmentalization strategies. We also have it as future work to pursue interesting compartmentalization strategies, building on HAKC to allow empirical comparisons.

VIII. Related Work

Prior work includes isolation solutions that effect almost all parts of the computing stack, ranging from hardware extensions to novel user-space abstractions.

A. Isolation in Computer Systems

Kernel security is a long-standing and ongoing research topic. Prior work includes creating and improving isolation domains in both microkernels [30] and monolithic kernels [18], [60], [62], [73], [75]. Non-monolithic kernels, as well as some monolithic isolation methods, require significant kernel redesign, while HAKC is not as intrusive. Furthermore, HAKC allows for fine-grained isolation, unlike some of the methods listed above. There is also work regarding isolation in user-space [7], [26], [28], [33], [52], [53], [59], [93]. However, these techniques often rely on kernel abstractions, hence they are not applicable for kernel isolation, or would require the introduction of a trusted software layer beneath the kernel, i.e., hypervisor. Much of the work to handle privilege separation and isolation can significantly affect performance, hence works like Split Kernel [43] have been developed to select the level of protection and isolation for kernel functions based on if a trusted process is utilizing kernel functionality. HAKC is an *always on* solution that protects against exploits even from trusted processes.

Another approach for operating system isolation are library OSes [22], [36], [46], [72], [77], which restrict the operating system exposed to applications. Built on library OSes, multiple works [14], [17], [42], [44], [54], [56], [83] have investigated unikernels — purpose-built kernels and user-spaces for a single application — and ways to create, improve, and use these minimalistic systems. Compared to HAKC, these approaches achieve isolation by separating kernel memory based on what each application needs. However, unlike the previous work, HAKC runs directly on bare metal, without any monitor reducing the trusted computing base. Furthermore, HAKC is much more flexible in defining different levels of granularity to allow for trade-offs between performance and security.

Finally, there have been research efforts in leveraging language properties to address memory related issues. Previous work uses Rust to implement operating systems [8], [61], [70] as well as unikernels [44], to utilize the language's type and memory safety to obtain isolation and increase security. While Rust prevents many memory related bugs, in order to prevent data-only attacks involving accessing valid, live memory areas, a compartmentalization system like HAKC is required.

B. Hardware Based Isolation

Intel's Memory Protection Keys (MPK) is an x86 extension that allows a process to partition memory into 16 domains. R/W privileges for each domain are then controlled by modifying a special key policy register, which is accessible from ring 3. User space access has motivated efforts to enforce isolation using MPK in a secure manner [31], [40], [71], [74], [78], [84]. Certain MPK-based isolation schemes are vulnerable to attacks that leverage kernel system calls to subvert MPK permissions [15]. Unlike HAKC, MPK is designed to provide coarse-grained, page level isolation. Intel's Software Guard Extensions (SGX) [34] provide another avenue for compartmentalization, however, SGX is intended for user-space enclave-like protection against a malicious operating system. SGX has also been shown to induce significant overhead [94].

Other works have focused on using hardware to support *safe regions* — regions of memory only accessible by privileged instructions — but have only extended simulated hardware and have focused on user-space applications [25], [58]. There are several works on using hardware tagging to support various compartmentalization and pointer bounds checking schemes [21], [39], [76], [80], [92], however most of these works are implemented on simulated architectures. One effort in particular, Mondrix [89], provides inter-modular Linux kernel compartmentalization using a 2-bit word granularity tagging extension [88]. Unlike HAKC, Mondrix is implemented in simulation, and requires a memory supervisor that monitors all kernel permission changes. Furthermore, Mondrix only implements inter-module isolation, whereas HAKC supports both inter-module and intra-module isolation. The Cheri project [90] provides architecture extensions to support pointer capabilities, which can be used to encapsulate memory. Cheri's fixed capability model provides less flexibility than PAC, where arbitrary information can be used as the context to sign pointers. Further, Cheri's focus on capabilities misses data-only attacks.

Arm TrustZone is a security feature on Armv8-A and Armv8-M [2] architectures that provides strict memory isolation between a privileged *secure world* and an unprivileged *normal world*. Lack of inter-process isolation between applications in the secure world as well as applications in the normal world have inspired isolation schemes that leverage the TrustZone ecosystem [9], [11], [85], [95]. Unlike HAKC, these systems focus on enforcing inter-process isolation and providing safe regions applications to store sensitive data.

C. Arm PAC and MTE Extensions

Recent works have utilized Arm PAC to enforce control-flow integrity (CFI), spatial memory safety, and code pointer integrity (CPI). PACStack [49] is a CFI scheme that secures return addresses stored on the stack through a chain of hashing, where a hash for each return pointer is unique based on the current execution path of a program. PTAuth [23] enforces temporal memory safety by storing a unique id at the base of data object, using the unique id as the PAC context during signing and authentication. PARTS [50] is an LLVM instrumentation framework that utilizes PAC to support a CPI scheme that is resistant to pointer-reuse attacks, and thwarts control-flow and data-oriented attacks. Compared to these schemes,

HAKC can provide wider protection against many classes of attacks, and in some cases, like with PACStack, can be used in conjunction. HAKC is the first design to the best of our knowledge that utilize MTE-based isolation. However, designs have been proposed that would leverage MTE-like architectural features to improve the Clang AddressSanitizer [12].

D. Isolation with Hypervisors

Monolithic kernels such as Linux are known to be vulnerable to faulty or malicious subsystems, such as device drivers and network stacks. This issue has motivated researchers to leverage hypervisors and virtualization schemes to isolate kernel subsystems [24], [60], [62], [64]. One example of a hypervisor-based solution is VirtuOS [64]. VirtuOS isolates various Linux kernel subsystems by using the Xen hypervisor to create service domains. Although efforts are made to reduce domain communication overhead, the copying of data, file descriptor translation, and the migration of domain-specific information incur significant overhead. Another example is HUKO [91], also based on Xen, which isolates untrusted extensions. Both these schemes include the hypervisor in the TCB, while HAKC relies on hardware for enforcement. There are multiple known vulnerabilities in existing hypervisors, and although work has been done to address this [47], [63], [79], verifying hypervisor implementations is a difficult task. Unlike hypervisor based isolation schemes, which focus on isolating systems at the granularity of kernel modules and subsystems, HAKC is capable of compartmentalizing *bare-metal* LKMs at a finer granularity, including compartmentalizing subsystems within an LKM.

IX. Conclusion

We present Hardware-Assisted Kernel Compartmentalization, which provides a practical and performant way to compartmentalize commodity monolithic kernels using a novel application of hardware features. HAKC provides strong protections for code and data pointers used in compartmentalized code, while minimizing performance overhead and developer effort. We are releasing all related code as open source at https://github.com/mit-ll/HAKC.

Acknowledgements

We thank Hui Peng and the anonymous reviewers for their detailed feedback that helped improve the paper. We also Bob Bond, Marc Zissman, and Roger Khazan whose support and guidance has made this effort possible. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), NSF CNS-1801601, and DARPA HR001119S0089-AMP-FP-034.

References

[1] The kernel address sanitizer (kasan). 2021.

[2] ARM. Trustzone.

[3] LTD. ARM. Pointer authentication on armv8.3, 2017.

[4] LTD. ARM. Armv8.5-a memory tagging extension, 2019.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[6] Roberto Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, Mar. 2017.

[7] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 550–564, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[9] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Network and Distributed Systems Security (NDSS) Symposium*, February 2019.

[10] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. Kaslr: Break it, fix it, repeat. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, page 481–493, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Dawei Chu, Yuewu Wang, Lingguang Lei, Yanchu Li, Jiwu Jing, and Kun Sun. Ocram-assisted sensitive data protection on arm-based platform. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 412–438, Cham, 2019. Springer International Publishing.

[12] Clang. Hardware-assisted addresssanitizer design documentation.

[13] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[14] Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, and Mohamed Kassi-Lahlou. Unikernel-based approach for software-defined security in cloud infrastructures. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018.

[15] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on pku-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020.

[16] Kees Cook. Kernel address space layout randomization. *Linux Security Summit*, 2013.

[17] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, page 36–41, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[19] Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. In *Proceedings of the 9th Annual International Conference on Computing and Combinatorics*, COCOON'03, page 351–363, Berlin, Heidelberg, 2003. Springer-Verlag.

[20] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J. E. Ekberg. Camouflage: Hardware-assisted cfi for the arm linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[21] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *SIGOPS Oper. Syst. Rev.*, 42(2):103–114, March 2008.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery.

[23] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[24] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[25] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, Baltimore, MD, August 2018. USENIX Association.

[26] TAL GARFINKEL, SHRAVAN NARAYAN, CRAIG DISSELKOEN, HOVAV SHACHAM, and DEIAN STEFAN. The road to less trusted code. *USENIX PATRONS*, page 15, 2020.

[27] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 179–194, 2016.

[28] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. Enclosure: Language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 255–267, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Google. syzkaller, 2016.

[30] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[31] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[32] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, page 113–126, USA, 1992. USENIX Association.

[33] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 393–405, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Intel. Intel® software guard extensions.

[35] 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. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[36] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[39] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.

[41] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. Portable software fault isolation. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 18–32, 2014.

[42] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. *Unikraft: Fast, Specialized Unikernels the Easy Way*, page 376–394. Association for Computing Machinery, New York, NY, USA, 2021.

[43] Anil Kurmus and Robby Zippel. A tale of two kernels: Towards ending kernel hardening wars with split kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1366–1377, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ştefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. Flexos: Making os isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 79–87, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. 2021.

[48] Hans Liljestrand, Zaheer Gauhar, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Protecting the stack with paced canaries. In *Proceedings of the 4th Workshop on System Software for Trusted Execution*, SysTEX '19, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[50] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 177–194, USA, 2019. USENIX Association.

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[52] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, Savannah, GA, November 2016. USENIX Association.

[53] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1023–1040, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, May 2015. USENIX Association.

[55] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[56] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, December 2013.

[57] Jonathan M. McCune, Adrian Perrig, Arvind Seshadri, and Leendert van Doorn. Turtles all the way down: Research challenges in user-based attestation. In *2nd USENIX Workshop on Hot Topics in Security (HotSec 07)*, Boston, MA, August 2007. USENIX Association.

[58] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. Microstache: A lightweight execution context for in-process safe region isolation. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 359–379, Cham, 2018. Springer International Publishing.

[59] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[60] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. Lxds: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association.

[61] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[62] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Tu Dinh Ngoc, Boris Teabe, Alain Tchana, Gilles Muller, and Daniel Hagimont. *Mitigating Vulnerability Windows with Hypervisor Transplant*, page 162–177. Association for Computing Machinery, New York, NY, USA, 2021.

[64] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACMSymposium on Operating Systems Principles*, SOSP '13, page 116–132, New York, NY, USA, 2013. Association forComputing Machinery.

[65] NIST. Cve-2016-4997 detail, 2021.

[66] NIST. Cve-2017-9074 detail, 2021.

[67] NIST. Cve-2019-14815 detail, 2021.

[68] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein. Finding focus in the blur of moving-target techniques. *Security Privacy, IEEE*, 12(2):16–26, Mar 2014.

[69] Hamed Okhravi. A cybersecurity moonshot. *IEEE Security & Privacy*, 19(3):8–16, 2021.

[70] Hamed Okhravi, Nathan Burow, Richard Skowyra, Bryan Ward, Samuel Jero, Roger Khazan, and Howard Shrobe. One giant leap for computer security. *Security Privacy, IEEE*, 2020.

[71] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[72] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 291–304, New York, NY, USA, 2011. Association for Computing Machinery.

[73] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577, 2020.

[74] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. Keeping Safe Rust Safe with Galeed. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC'21)*, Dec 2021.

[75] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, André DeHon, Jonathan M Smith, et al. μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In *24nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2021)*. USENIX Association, 2021.

[76] Nick Roessler and André DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 478–495, 2018.

[77] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[78] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[79] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

[80] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2016.

[81] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. Glitching demystified: Analyzing control-flow-based glitching attacks and defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 400–412, 2021.

[82] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. *Conware: Automated Modeling of Hardware Peripherals*, pages 95–109. Association for Computing Machinery, New York, NY, USA, 2021.

[83] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[84] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[85] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery.

[86] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: a framework for virtualization-based security systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, Santa Clara, CA, July 2015. USENIX Association.

[87] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS X, page 304–316, New York, NY, USA, 2002. Association for Computing Machinery.

[89] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, New York, NY, USA, 2005. Association for Computing Machinery.

[90] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, 2014.

[91] Xi Xiong, Donghai Tian, and Peng Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, 2011.

[92] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.

[93] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[94] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. On the performance of intel sgx. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187, 2016.

[95] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Minimal kernel: An operating system architecture for TEE to resist board level physical attacks. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 105–120, Chaoyang District, Beijing, September 2019. USENIX Association.