LatticeBox: A Hardware-Software Co-Designed Framework for Scalable and Low-Latency Compartmentalization

Zhanpeng Liu*†, Chenyang Li*†, Wende Tan§, Yuan Li¶✉,
Xinhui Han*‖✉, Xi Cao‖, Yong Xie‡, Chao Zhang†¶‖
*Wangxuan Institute of Computer Technology (WICT), Peking University
†Institute for Network Sciences and Cyberspace (INSC), Tsinghua University,
¶Zhongguancun Laboratory, §Imperial College London, ‡Qinghai University,
‖JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

***Abstract*—Modern software systems increasingly rely on compartmentalization to isolate untrusted or potentially vulnerable components, such as third-party drivers and JIT-compiled code. However, existing hardware isolation techniques suffer from scalability constraints, high switching latency, and inadequate security guarantees. In particular, permission-changing instructions used by some compartmentalization technology, such as Intel MPK's `WRPKRU`, can be exploited by untrusted code, which complicates the secure deployment process.**

In this paper, we introduce LatticeBox, a hardware-software co-designed framework that addresses these limitations using a lattice-based access control model. LatticeBox encodes permissions and memory regions as compact, hierarchical N-bit vectors. This design enables a hardware architecture that reduces domain-switching latency to a single CPU cycle and inherently prevents misuse of permission-switching instructions. Additionally, LatticeBox employs a customized instruction (`lp_land`) to enforce strict cross-domain control-flow integrity, effectively preventing unauthorized indirect function calls. We implement LatticeBox on a RISC-V BOOM core and evaluate it using both microbenchmarks and real-world applications, including WebAssembly runtimes and Linux kernel modules. Our results show that LatticeBox achieves domain switching up to 180× faster than Intel MPK while supporting fine-grained, scalable isolation. Evaluation on real-world workloads demonstrates only a modest performance impact, with only 2% slowdown for enhanced WebAssembly runtimes and just 3% lower throughput for ApacheBench running isolated Linux kernel modules.

I. Introduction

Modern software systems are increasingly dependent on isolation mechanisms to securely execute untrusted or potentially vulnerable components. In web browsers, sandboxing is crucial to confine JavaScript and WebAssembly execution, as well as to isolate components like image or video codecs that are frequently targeted by exploits [1], [2], [3]. Operating systems, especially those based on microkernel architectures, isolate device drivers and kernel services to minimize the trusted computing base (TCB) [4], [5], [6]. Similarly, serverless computing platforms and multi-tenant web services rely on isolation to protect per-user data [7], [8]. Reliable fine-grained isolation is thus indispensable, yet current mechanisms impose difficult trade-offs between security, performance, and scalability. For decades, isolation has relied on segmentation and paging. Segmentation is now deprecated [9], while paging-based schemes incur performance penalties from frequent privilege-level transitions and TLB flushes [10]. Other approaches such as page-table-switching or Software Fault Isolation (SFI) [11], [12], [3] incur high runtime or memory overhead and often require compiler-level instrumentation or binary rewriting, limiting their practicality and compatibility with legacy systems.

However, more recent hardware-assisted isolation mechanisms have attempted to overcome these shortcomings. Among these mechanisms, Intel MPK [9] is widely used due to its lower performance cost and its ability to efficiently support intra-address space isolation without requiring significant modifications to existing hardware architectures or software toolchains. Nevertheless, MPK itself suffers from several critical limitations [13], [14]. Firstly, its design only permits sixteen protection domains due to the two-bit allocation per key in the PKRU register. This scale is insufficient for browsers or serverless runtimes, which typically require hundreds of compartments. Attempts to raise this limit, such as EPK [15], VDom [16], Donky [17] and BULKHEAD [18], rely on a two-level isolation scheme which increases the number of keys. However, each additional indirection lengthens a switch and latency grows rapidly as the domain count rises. Secondly, the MPK security model is weakened by the unprivileged permission-switching instruction (i.e. the `WRPKRU` instruction), which allows user code to modify permissions at will. Practical defenses require binary scanning and rewriting to prevent sandboxed code from executing such instructions. However, this introduces execution latency, which is partic-

✉Corresponding authors: lydorazoe@gmail.com, hanxinhui@pku.edu.cn

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240515
www.ndss-symposium.org

ularly fragile for JIT scenarios. Thirdly, every permission change must be followed by a serialising memory barrier, which introduces delays of hundreds of cycles and renders MPK unsuitable for workloads involving frequent switching. The limitation has led high-performance engines such as V8 to avoid MPK for sandbox isolation [19].

To overcome these shortcomings, we introduce LatticeBox, a hardware–software co-design framework that replaces MPK-style key lookup with the principled Lattice-Based Access Control (LBAC) model [20]. In LBAC, both code compartments and data pages are labelled with an N-bit vector drawn from a Boolean lattice, and memory access is permitted only when the execution label dominates the target label.

This representation encodes permissions and domains within the same space, yielding three key benefits. ❶ **Scalable isolation:** It removes the scalability limitations of traditional Access Control Lists (ACLs) by reducing metadata size from linear to O(log N) relative to the number of system domains. For instance, with a 10-bit label, it can support hundreds of mutually isolated domains. ❷ **Single-cycle domain switches:** Permission checks and updates become simple local bitwise operations on compact bit sets, enabling architectures where permissions are directly tied to pipeline instructions. This eliminates memory barriers during context switches, reducing latency to a single cycle. ❸ **Secure-by-design:** Since privilege escalations are expressed solely as bounded label updates performed by a dedicated instruction, there is no unprivileged analogue to the `WRPKRU` instruction. Consequently, LatticeBox is immune to gadget-style permission forgery and does not require binary scanning. Labels are stored in page-table entries and cached in the TLB, adding only a few tag bits of metadata. Meanwhile, the `lp_land` instruction, a minimal ISA extension, atomically raises privilege and validates cross-domain control transfers. *These features, together, provide scalable, single-cycle, and security-hardened compartmentalization at negligible hardware cost.*

We implement a prototype of LatticeBox on a RISC-V BOOM [21] core and evaluate it using both synthetic microbenchmarks and real-world workloads, including SPEC CPU 2006 running on an enhanced WebAssembly runtime (wasm2c) and Apache benchmark tests with isolated IPv6 Linux kernel modules. Experimental results demonstrate that LatticeBox achieves domain switching up to 180× faster than Intel MPK while maintaining near-native performance in practical deployments: the enhanced wasm2c runtime exhibits only 2% runtime overhead, and the isolated IPv6 modules reduce Apache server throughput by just 3%. Furthermore, LatticeBox supports an order of magnitude more compartments than conventional approaches while providing stronger isolation guarantees through its robust security semantics.

In summary, this paper makes the following contributions:

- We introduce Lattice-Based Access Control (LBAC) model into memory access that scales to thousands of domains while eliminating access-control lists and key virtualization requirements.
- We design a lightweight instruction set architecture (ISA) and microarchitectural extension that implements LBAC with single-cycle domain switches and built-in cross-domain control-flow integrity, while requiring only minimal changes to existing operating systems and toolchains.
- We prototype the full stack and conduct a comprehensive evaluation against state-of-the-art isolation schemes, demonstrating that strong security guarantees can be achieved with negligible performance and hardware cost.

II. Background

A. WebAssembly

WebAssembly (Wasm) [22] is a binary instruction format designed to execute code at near-native speeds within web browsers, providing a secure and efficient platform-independent runtime. It has become a popular method for isolating sensitive or potentially vulnerable modules in web applications due to its strong sandboxing capabilities and lightweight execution environment [3].

However, WebAssembly isolation also has notable limitations. WebAssembly provides efficient sandboxing, especially in the 32-bit wasm32 mode, where guard pages can be used to enforce memory isolation with minimal performance overhead. However, this model is fundamentally limited to a 4 GB linear memory space due to its 32-bit address space. More critically, the safety guarantees rely heavily on the correctness of the compiler and runtime [23]. A miscompilation or validation bug can allow memory accesses to escape the intended sandbox. With the introduction of wasm64 or support for multiple linear memories [24], guard pages are no longer sufficient, and explicit bounds checking must be inserted. This adds significant runtime overhead and can degrade performance, especially in memory-intensive applications.

B. Monolithic Kernels and Microkernels

Operating system kernel architectures can be broadly classified into monolithic kernels and microkernels [25]. Monolithic kernels, such as Linux, integrate core functionalities—including device drivers, file systems, and network stacks—directly into the kernel space. This design offers high performance and low communication overhead between kernel components. However, it also results in a large trusted computing base (TCB), which increases the attack surface. In practice, many security vulnerabilities have been found in kernel subsystems such as device drivers and networking stacks, making them common targets for privilege escalation attacks. In contrast, microkernels aim to minimize the kernel's responsibilities by moving most services, such as drivers and network protocols, to user space. This architectural separation reduces the TCB and can improve fault isolation. Nevertheless, the performance cost of frequent inter-process communication (IPC) between user-space services and the microkernel remains a significant challenge [26]. These IPC costs are especially problematic in high-throughput or latency-sensitive scenarios, limiting the practical adoption of microkernels in performance-critical systems.

C. Intel MPK

Intel Memory Protection Keys (MPK) is a hardware feature that provides efficient and flexible memory access control at the user level. It repurposes unused bits in page table entries (PTEs) to embed memory keys that distinguish different data regions in memory. These keys are used as indices into a special CPU register called PKRU (Protection Key Rights for User pages, and PKS for kernel space [9]), which defines the access rights for each region. Each thread can change its own permissions using the `WRPKRU` instruction. This does not require modifying page tables or invoking the operating system, making it significantly faster than traditional isolation schemes. Despite its advantages, MPK has several limitations. Its scalability is constrained by the metadata used for access control, which grows linearly with the number of supported domains. Currently, MPK supports only 16 protection keys, making it unsuitable for applications that require many isolation domains, such as web browsers or serverless platforms. Performance is also affected. `WRPKRU` instruction updates PKRU register with values drawn from general-purpose registers. While this design offers flexibility, it negatively impacts instruction pipeline efficiency. Since memory access permissions change with each PKRU update, the processor must stall the pipeline to wait for the update to complete before executing subsequent memory access instructions. This restriction prevents out-of-order execution, introducing noticeable pipeline latency on the order of hundreds of CPU cycles. Security is also a concern: the `WRPKRU` instruction is unprivileged and executable in user space, allowing untrusted code to arbitrarily change permissions. Preventing such misuse typically requires static binary scanning and rewriting to ensure that isolated code contains no hidden `WRPKRU` instructions, which complicates deployment in adversarial environments.

D. Lattice-Based Access Control (LBAC)

Lattice-Based Access Control (LBAC) [27], [28] leverages the lattice structure to enforce strict, hierarchical security policies. In LBAC, subjects (e.g., processes or users) and objects (e.g., memory pages or files) are labeled with security labels organized into a lattice. Access is permitted if and only if the subject's label dominates the object's label within the lattice. This dominance reflects the partial order relation intrinsic to lattices. The lattice structure also supports operations like meet ($\wedge$, greatest lower bound) and join ($\vee$, least upper bound), which are useful for defining shared access. For example, if two subjects, A and B, both need access to an object, the object's label can be set to the meet ($\wedge$) of A and B's labels to ensure it remains accessible to both while preserving containment guarantees.

Despite these properties, LBAC has historically received limited attention in the computer systems community and has rarely been applied to memory isolation or protection contexts. Traditional memory access management approaches commonly rely on Access Control Lists (ACLs), which scale poorly with increasing numbers of isolated domains. In this work, we identify that lattice structures offer unique advantages for memory isolation: they can compactly encode hierarchical permissions using concise N-bit vectors, allowing rapid permission checks through simple bitwise comparisons. This characteristic enables exceptionally efficient hardware implementations, significantly reducing the complexity and overhead associated with managing permissions at runtime.

III. Motivation and Threat Model

LatticeBox is motivated by the observation that the performance and scalability bottleneck of Intel MPK stems from its ACL-style design: permissions are stored in dedicated registers that must be updated and serialized, complicating hardware, increasing switching overhead, and creating misuse opportunities. We adopt a Lattice-Based Access Control (LBAC) model for memory protection, where permissions are encoded as compact, hierarchical N-bit vectors matching memory region lengths. This design reduces domain-switching latency to a single CPU cycle, prevents misuse of permission-switching instructions and Spectre-v1 attacks, and offers scalability.

We adopt a threat model that is at least as strong as those used in most MPK-based solutions and aligns with the native isolation model of HFI. We assume an attacker capable of executing arbitrary code, including self-modifying code, within a designated compartment. This model applies to both user-space and kernel-space contexts. The attacker may attempt to exploit memory vulnerabilities—such as buffer overflows or use-after-free bugs—to escalate privileges or access unauthorized memory within or beyond the compartment. However, the attacker's ability to read or write memory is strictly confined by the hardware-enforced virtual memory protection model. Specifically, the system ensures that the attacker cannot access memory regions outside the permitted domains, as defined by the LBAC policy.

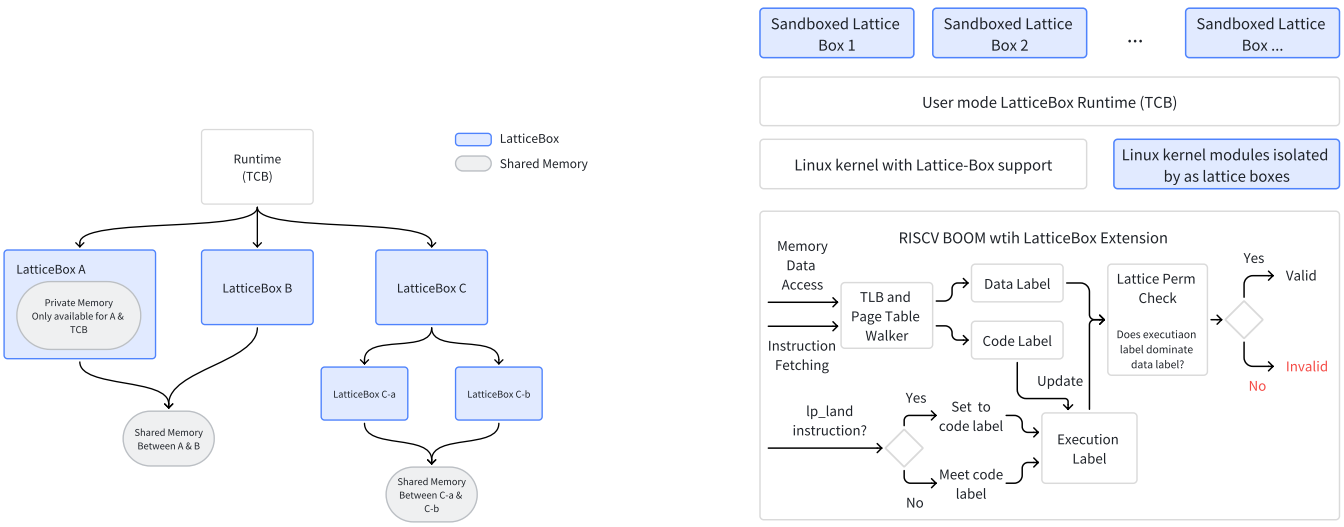
We consider the following attacks out of scope: 1) Physical-level attacks, including Direct Memory Access (DMA)-based probing; 2) Microarchitectural side channels, such as cache timing or speculative execution except Spectre-v1; 3) Denial-of-service (DoS) attacks, which target system availability rather than isolation guarantees.

The goal of this work is to ensure that *even a fully compromised compartment cannot breach the confidentiality or integrity of other compartments, under the enforcement of LBAC at the hardware level.*

IV. Design

A. System Overview

LatticeBox is a hardware-software co-designed system that provides fast, scalable, and secure memory compartmentalization through a Lattice-Based Access Control (LBAC) memory access model. At the core of LBAC is the concept of a lattice, which is a partially ordered set equipped with two fundamental binary operations: meet ($\wedge$), representing the greatest lower bound, and join ($\vee$), representing the least upper bound. This mathematical structure is particularly well-suited for modeling hierarchical, sparse, and composable access relationships.



(a) Isolation Model

(b) Architecture Overview

Fig. 1: The overview of LatticeBox design.

As illustrated in Figure 1a, LatticeBox supports the creation of multiple "lattice boxes", which are specialized sandboxes with isolated memory regions. Each lattice box has private memory that is accessible only to itself and to compartments with a higher label in the lattice hierarchy, such as the trusted computing base (TCB). It is protected from access by other lattice boxes that do not have a dominating label. To facilitate inter-sandbox communication, lattice boxes can share memory corresponding to their meet ($\wedge$) in the lattice, ensuring that only mutually authorized interactions occur.

Figure 1b presents the system overview. The underlying hardware enforces access control based on labels during memory access. Above this, the operating system kernel manages the virtual memory system and, consequently, the associated labels. Any code that does not require direct management of virtual memory can be moved out of the core kernel and executed within a lattice box, such as Linux kernel modules. At the user level, a user-mode lattice box runtime allows multiple lattice boxes to execute within a shared address space, while enforcing strict and controlled memory access patterns.

To make the labels enforceable in hardware, three minimal hardware extensions are introduced. First, page-table entries and TLB entries are extended to include the labels. For data TLB, the label specifies the required permission level to access the corresponding memory region. Second, each CPU core maintains a dedicated control register that holds the current execution label. On every memory access, the system checks whether this label dominates the label of the accessed memory. If not, a custom exception is raised. In the instruction TLB, the label represents the upper bound on the permission level of the code. When control flow enters a new page with a different label, the core's execution label is lowered to the meet ($\wedge$) of the previous label and the new page's label. Third, LatticeBox introduces a new instruction, `lp_land`, to securely manage cross-domain transitions. When control is transferred between compartments, `lp_land` verifies the validity of the transition and updates the core's execution label to match the label stored in the instruction TLB entry of the target page. This mechanism supports controlled permission escalation. The `lp_land` instruction eliminates the need for expensive memory fences and prevents unauthorized privilege escalation by design, which addresses a critical weakness of MPK's unprivileged instruction model.

The software layer of LatticeBox requires minimal modification to existing systems. The operating system merely assigns permission labels when mapping pages into memory and manages stack transitions when crossing privilege boundaries. Existing programs and runtimes, including dynamically generated JIT code, remain fully compatible after minor adjustments to memory allocation calls. The result is a practical, drop-in approach to secure compartmentalization that requires no invasive binary scanning, rewriting, or extensive toolchain modifications. In summary, LatticeBox significantly surpasses prior isolation technologies such as Intel MPK by addressing their key limitations explicitly: **(1) Low Latency:** Single-cycle compartment transitions replace MPK's costly memory fences, reducing latency from hundreds of cycles to just one cycle. **(2) Scalability:** Compact lattice labels support billions of compartments, eliminating MPK's severe limit of sixteen domains. **(3) Security:** Built-in cross-domain control-flow integrity enforced by the dedicated `lp_land` instruction inherently prevents gadget-style permission forgery without expensive binary rewriting.

By combining these novel architectural strategies with minimal hardware and software modifications, LatticeBox provides a practical, secure, and highly performant compartmentalization framework suitable for modern software environments.

B. Lattice-Based Access Control in Memory

Traditional memory access permission mechanisms often rely on Access Control Lists (ACLs) [29], [13], [30], which can incur substantial complexity during permission management—particularly when updating or verifying access rights. This complexity grows linearly with the number of memory regions and access relationships, making such designs non-scalable. However, in practice, the memory access patterns in many systems are highly sparse. Most code modules require access to only a small subset of all available memory regions. This sparsity is especially evident in sandboxed execution environments. In a typical setup, a trusted host runtime (i.e., external code) requires broad access to every sandbox's memory, whereas each untrusted sandbox module (i.e., internal code) only accesses its own isolated memory region. In a system with N sandboxes, the effective permission model boils down to just two meaningful relationships:

- The host has access to all sandbox memory regions.
- Each sandbox has access only to its own memory region.

Despite this simple structure, encoding it via ACLs requires storing $N + 1$ entries—one for the host and one for each sandbox. Worse yet, any update to code or memory requires revisiting and potentially modifying this list, introducing inefficiencies and increasing the chance for error.

One potential simplification involves encoding permissions using a global "host" privilege along with one privilege per sandbox. This compact representation uses approximately $\log_2 N$ bits per permission and is sufficient to express the desired access control relationships for N sandboxes. However, this scheme eliminates the possibility of memory sharing between non-host components. Since non-host modules cannot directly share memory, any inter-sandbox communication must be routed through the host using memory copies—a process that incurs considerable overhead when communication is frequent or performance-critical.

Figure 1 illustrates how we model memory access permissions with lattice structure. In our model, both code and memory regions are assigned permission levels drawn from the same lattice. A code component with permission level a is allowed to access a memory region with permission level b if and only if $a >= b$ in the lattice ordering. This single-rule access control model enables a simple yet expressive framework for enforcing permissions.

We designate the host runtime as part of the system's trusted computing base (TCB). Because it must access all memory regions, we assign it the top element of the lattice—formally, the join ($\vee$) of all other permission levels. This guarantees that the host can access any memory region without exception. Sandboxes in this model can be made mutually isolated by construction: their permission levels are incomparable in the lattice (i.e., there is no ordering between sandbox i and sandbox j when $i \neq j$). This ensures that no sandbox can directly access another's memory. However, if communication is required, it can be achieved through memory regions placed at the meet ($\wedge$) of the corresponding permissions. These shared regions act as controlled junctions for data exchange. Because these sandboxes are isolated based on lattice-encoded permissions, we refer to them as "lattice-boxes".

Beyond simple isolation, lattice-boxes naturally support nesting and composition, thanks to the hierarchical nature of the underlying permission lattice. A lattice-box can encapsulate other lattice-boxes, each inheriting a constrained subset of the parent's permissions, thereby enabling modular organization and fine-grained delegation of access. Likewise, multiple lattice-box systems can be composed into a larger system by ensuring their upper bounds remain distinct, while the host runtime is granted access via the join ($\vee$) of all participating subsystems' permissions. This structural flexibility makes lattice-boxes well-suited for a wide range of compartmentalization patterns, from flat isolation schemes to deeply nested or collaborative module arrangements.

C. Architecture Design

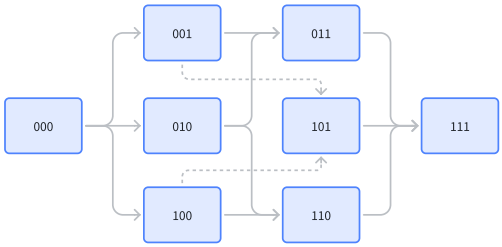


Fig. 2: Subset Lattice with Bit Vector Encoding.

1) Bit Vector Encoding: To support efficient implementation—especially in hardware—we encode permission levels as bit vectors. The lattice ordering is defined such that $a >= b$ if and only if every bit in a is greater than or equal to the corresponding bit in b. As shown in figure 2, this encoding naturally forms a subset lattice (also known as a Boolean lattice), which offers exponential scalability in creating mutually isolated lattice-boxes. For example, using a 10-bit encoding, we can select all vectors with exactly five bits set. These vectors are pairwise incomparable, and thus define $C(10, 5) = 252$ mutually isolated lattice-boxes.

However, introducing shared memory between sandboxes reduces the total number of mutually isolated boxes that can be supported. This is because the meet ($\wedge$) of two permissions may inadvertently intersect with other sandboxes' meets or permission levels. For example, in a 4-bit encoding system, the meet ($\wedge$) of 1100 and 0011 reaches the lowest bound. In the worst-case scenario, where all possible communication patterns must be supported, the number of supported lattice-boxes scales linearly with the number of bits, rather than exponentially. However, such dense communication requirements are rare in practice; most compartmentalization systems require only localized or structured sharing.

To balance isolation and flexibility, the bit vector can be partitioned to support nesting and composition. For example, a 10-bit vector can be split into two segments:

- The first 5 bits define $C(5,2) = 10$ outer lattice-boxes, which are fully isolated from one another.
- The remaining 5 bits define 5 nested lattice-boxes within each outer box, supporting arbitrary memory sharing among them.

For N bits encoding, LatticeBox supports at most $C\left(N, \lceil \frac{N}{2} \rceil\right)$ isolated lattice-boxes. If memory sharing is needed, split N into $M + K$, supporting $C\left(M, \lceil \frac{M}{2} \rceil\right)$ outer lattice-boxes and K nested ones, totaling $K * C\left(M, \lceil \frac{M}{2} \rceil\right)$ lattice-boxes with elaborated access permissions. This approach provides a scalable, structured mechanism to support both strict isolation and efficient communication. In contrast, MPK's ACL-based design has scalability limits: the ACL size doubles with each added label bit. For example, at n=10, Intel MPK supports 1024 memory regions but needs a 2048-bit PKRU register, which is hard to maintain.

2) Memory Data Permission Level Management: LatticeBox embeds lattice permission keys (LPKs) into unused PTE bits, which is similar to Intel MPK but with richer semantics. LPKs encode permission levels directly, eliminating the need for external lookup structures. While this richer encoding requires more bits per key, a 10-bit vector is feasible across major architectures, including x86, ARM, and RISC-V. To support more bits, alternatives include reallocating part of the physical address space for permission metadata or extending the page table, such as by adding a shadow table for larger encodings.

On each memory access at the data cache, LatticeBox performs an additional permission check alongside the standard read/write validation based on PTE flags. It verifies whether the processor holds a permission level that dominates the memory's label in the lattice. If the check fails, a custom exception is raised to prevent unauthorized access.

To preserve backward compatibility and enforce secure defaults, we define the all-zero bit vector as the top of the permission lattice—the most privileged level. This ensures that, by default, only code running at the highest privilege level can access untagged or default memory regions. As a result, we adopt a counterintuitive but intentional ordering: in our lattice, a bit value of 0 is greater than 1. This inversion aligns with conservative security principles, enabling safe defaults while supporting expressive and scalable access control.

3) Core Permission Management: To reduce permission switching latency and provide enhanced security, we propose a permission control mechanism that leverages LPKs embedded in the PTEs of executable pages to restrict processor execution rights. As illustrated in figure 3, when the processor dispatches an instruction, it checks whether the instruction originates from an executable page with a different LPK. If so, it computes the meet ($\wedge$) between the current permission state and the LPK of the new page, reducing access rights accordingly. Controlled permission escalation is achieved through a special instruction, `lp_land`, which can optionally escalate the lowered permission. Crucially, the escalation is strictly bounded by the LPK of the page containing the `lp_land` instruction. The instruction is named to reflect its intended

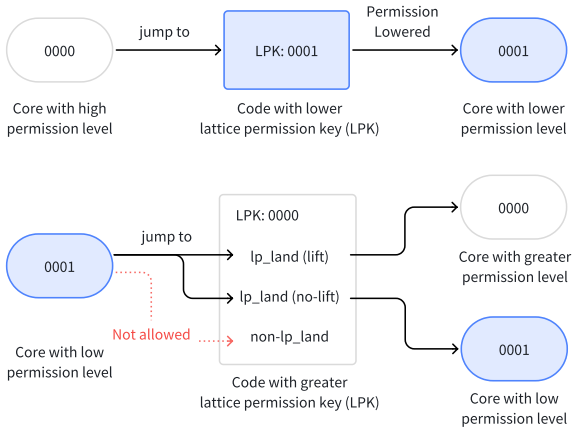


Fig. 3: Core Permission Management.

role as the entry point for cross–lattice-box invocations. In addition to managing permission escalation, `lp_land` also records the caller's permission level, enabling the callee to adapt its behavior based on the origin of the call.

This approach offers several advantages. At the microarchitectural level, it allows permission transitions to occur during instruction dispatch, fully decoupled from the execution pipeline. As a result, permission changes no longer block out-of-order execution, significantly reducing latency. For permission lowers, even no extra instruction is actually executed. Moreover, this design strengthens security guarantees: even if a binary contains `lp_land` instructions, it cannot elevate permissions beyond what is permitted by the code segment's PTE. As long as potentially untrusted code is mapped with a low-permission LPK, it is confined by default.

A simpler alternative would be to treat the LPK in each executable page as the sole determinant of execution permission. However, this rigid model makes it difficult to support shared code across different compartments or threads with varying privileges. To address this, LatticeBox separates the processor's current execution permission from a code segment's upper-bound permission. This separation enables safe and efficient code reuse. For example, commonly reused library functions like memset or memcpy can reside in high-permission regions without performing permission escalation on entry. When executed from a lower-permission context, they run with reduced privileges, automatically respecting the caller's restrictions.

Furthermore, this model naturally extends to thread-level isolation. Each thread can be assigned its own thread-specific permission level, and the effective access permission is determined as the meet ($\wedge$) of the executing code's permission and the thread's assigned level. The hardware should keep thread permissions immutable in user space, ensuring strong isolation across threads without compromising flexibility or security.

4) Reduced Privilege for Non-TCB Code: While the proposed permission management scheme effectively restricts permission escalation via the `lp_land` instruction, it does

not eliminate all potential vectors for privilege abuse. A binary may still contain other instructions that can compromise system security. For instance, in user mode, an attacker could invoke system calls to escape sandbox constraints. In kernel mode, more severe threats exist—for example, an attacker might directly modify control status registers such as CR3, which holds the base address of the page table, thereby bypassing the permission model entirely.

To address these residual risks, LatticeBox reduces each processor mode's privilege based on the current lattice permission level. Certain ISA-level sensitive instructions, like syscalls or machine state change instructions, can only execute at the highest permission. Their classification is inherent to the ISA design, requiring no manual annotation. This ensures sensitive transitions or privileged operations are gated to the TCB, enabling safe execution of arbitrary binaries.

5) Cross Compartment Control Flow Integrity: Beyond direct memory access attempts, attackers can also manipulate control flow to launch attacks. Specifically, we must consider these two attack approaches:

- If an attacker can jump into the middle of a privileged function, they may exploit indirect calls within that function to escalate privileges. For example, the attacker could redirect an indirect call to a high-permission service function and manipulate callee-saved registers. This allows the attacker to re-enter the privileged function using attacker-controlled register context, and execute sensitive code with elevated permissions.
- Another related threat arises when a low-permission compartment calls a privileged function using a forged return address. Once the privileged function completes, it may return to an attacker-controlled address, resuming execution with a tampered register context.

To prevent these attacks, LatticeBox must enforce that low-permission lattice boxes may only enter service functions in higher-permission compartments through pre-defined entry points. And, the service functions should not easily trust the return address provided by the caller.

We extend the semantics of `lp_land` to solve these issues. Specifically, we require that: (1) if the jump occurs from a code segment whose permission is not greater than or equal to the destination's, then the first instruction at the target must be `lp_land`; and (2) the instruction immediately preceding `lp_land` must be a call instruction that explicitly writes the return address. This ensures that all permission-escalating transitions are intentional and verifiable, with `lp_land` acting as both the semantic and structural gatekeeper.

With the above design, LatticeBox enforces strict privilege escalation control by restricting permission changes exclusively to `lp_land` instructions. These instructions serve as secure gatekeepers at the entry points of designated high-permission service functions, ensuring full visibility of call origins to prevent confused deputy attacks. Specific software-based strategies to protect processor registers during these transitions are discussed in Section IV-D.

D. Software Design

To ensure secure compartment transitions, the software layer creates lattice boxes and manages secure and efficient transitions between them, into the runtime or across components operating under different permission levels. This leverages the hardware features provided by LatticeBox.

The first step in creating a lattice-box is to place the code requiring isolation into a memory region marked with a designated permission level. For precompiled binaries, these permissions can be applied at load time or immediately afterward. In the case of JIT-compiled code, the runtime can allocate a writable and executable memory region applied with appropriate permissions, and then populate it with generated code. To execute within a lattice-box, the module must not reuse the runtime's stack. Instead, a separate stack with the same permission level as the code must be initialized. A dedicated heap is also required for memory allocation. However, because the `lp_land` instruction records the caller's permission level, a shared runtime allocator can safely manage all heap memory by dynamically provisioning permission-specific ranges on demand.

When invoking an untrusted module (i.e., one with lower or non-comparable permissions), the caller must switch to a new stack, since the callee cannot operate on the caller's stack. Before switching, the caller must save all callee-saved registers to prevent them from being tampered with. Optionally, registers may also be cleared to avoid leaking sensitive data into the untrusted context. A stack switch is also required when invoking a trusted module. In this case, it is not necessary to protect the caller's registers, but the system must ensure the trusted module is protected from tampered stack pointers or forged return addresses supplied by an untrusted caller.

This design can sandbox untrusted code at the binary level without instrumentation. For trusted code, trampoline functions can be introduced to handle stack switches and register protections, remaining binary compatible. To minimize performance overhead, we recommend recompiling trusted code with minor modifications for direct lattice-box transitions.

V. Implementation

A. Processor Core

We extend the BOOM (Berkeley Out-of-Order Machine) core to support LatticeBox. BOOM is an open-source RISC-V processor that implements out-of-order execution, providing a more realistic model of pipeline behavior compared to in-order cores such as Rocket. While our prototype targets RISC-V, the LatticeBox design is architecture-agnostic and can be implemented on any platform that supports virtual memory and page tables. RISC-V was chosen primarily for its openness and the maturity of its surrounding ecosystem.

1) Page Table Walker and TLB Extensions: We modified the Page Table Walker (PTW) and TLB to support LPKs. During page table walks, LPKs are read from PTEs and stored in TLB entries. On TLB hits, the response includes LPKs for permission management. We use the 7 reserved bits in RISCV

PTEs and repurpose two PPN bits plus the top bit for a 10-bit LPK, preserving about 2 PB of addressable memory.

2) Fetch and Instruction Dispatch Modifications: During fetch, LPKs from the instruction TLB are transmitted with the instruction packet and attached as metadata during dispatch. Although BOOM supports out-of-order execution, instructions are dispatched in-order to the Re-Order Buffer (ROB), which allows us to focus on permission updating at dispatch time, correctly set the code permission for instruction, and do not invade into the execution pipeline.

In BOOM, CSR instructions are treated as "unique" and require exclusive access to the execution pipeline. If code permissions were updated using CSRs, frequent changes would degrade concurrency. To avoid this, we introduce a microarchitectural register, reg_eclp, which tracks the last dispatched instruction's effective code permission. During dispatch, each instruction's permission is computed using previous instruction's permission and the new instruction's LPK, following Equation 1 for regular instructions and Equation 2 for the special `lp_land` instruction. The result is used to update reg_eclp for the next instruction.

$$P_{new} = P_{pre} \wedge \ LPK_{new} \tag{1}$$

$$P_{new} = (P_{pre} \ \wedge \ LPK_{new}) \vee (Imm_{new} \ \wedge \ LPK_{new}) \tag{2}$$

In addition, the `lp_land` instruction is converted to an addi instruction at dispatch time, which avoids introducing additional pipeline complexity while enabling software to track the pre-jump permission level.

These design choices minimize the overhead of cross-domain control flow. Permission reduction occurs directly at the jump instruction itself, incurring no additional latency. When permission escalation is required, it behaves identically to an addi instruction in terms of execution cost.

3) Branch Misprediction and Exception Handling: BOOM supports branch prediction, and on a misprediction, the processor must revert to the pre-branch permission state. When a misprediction occurs, the pipeline is flushed, and reg_eclp is restored to the permission level associated with the mispredicted branch instruction. In the case of exceptions (traps or interrupts), the pipeline is also flushed, and control is transferred to the exception handler, which begins execution in kernel mode. To support this transition, the current value of reg_eclp must be saved to a CSR (CSR_LP_U for user mode and CSR_LP_S for kernel). We then set reg_eclp to 0, the lattice upper bound, allowing the exception handler to execute with maximum privilege. Upon returning from the exception, reg_eclp is restored from the saved CSR value.

4) Memory Access Module: During memory access, the system verifies whether the instruction has sufficient permission to access the target memory region. The minimum required permission corresponds to the LPK assigned to the memory page. Determining the processor's effective permission is context-dependent. In user mode, it is computed as the meet ($\wedge$) of the instruction's permission and the thread's permission. In kernel mode, the behavior differs based on the type of memory being accessed. If the kernel accesses a user page, the effective permission is the meet ($\wedge$) of the permission stored in CSR_LP_U and the thread's permission. If it accesses a kernel page, the effective permission is simply the instruction's permission. Because this check is performed concurrently with standard page table validations, it introduces no additional overhead. Moreover, this design naturally enforces privilege isolation during system calls—calls made from a low-permission context cannot be exploited to access high-permission memory, thus preserving security boundaries.

B. Kernel

We modify the Linux kernel to support embedding LPKs into PTEs and to manage the saving and restoring of lattice-permission-related CSRs. For CSR_LP_U, which tracks user-mode permissions, the kernel must save and restore its value during context switches between user tasks. Signal handling presents a special case, as it creates a new control flow and execution context. To maintain compatibility with the existing signal handling mechanism, we require the signal handler to manage the permission state of the interrupted control flow. As a result, the signal handler must be part of the runtime's trusted code. When delivering a signal, the kernel copies the value of CSR_LP_U into the signal handling context and resets CSR_LP_U to 0, effectively elevating the signal handler to the runtime's full permission level. For CSR_LP_S, which tracks previous kernel-mode control flow permission, the kernel must handle nested exceptions correctly. To support this, the kernel saves and restores CSR_LP_S to and from the trap frame on the kernel stack during exception handling.

VI. Case Study

A. WebAssembly Runtime

WebAssembly runtimes typically assume that the stack is safe and rely on guard pages to detect overflows in linear memory. However, this approach implicitly trusts the compiler to generate safe code. If the compiler contains vulnerabilities, it may emit code that accesses memory outside of the intended stack or linear memory regions. Moreover, guard pages are only effective for the 32-bit WebAssembly (wasm32) model. In the 64-bit variant (wasm64), explicit memory bounds checking is required, which can significantly degrade performance.

LatticeBox offers a more secure and efficient alternative. By placing the compiled WebAssembly code into a lattice box, we can isolate it from the rest of the system even if the compiler is buggy. This containment ensures that wasm code cannot access memory outside its designated region, reducing the reliance on code validation at load time and thereby improving startup performance. Additionally, since LatticeBox enforces memory isolation at the hardware level, it eliminates the need for software-based bounds checks, even in the wasm64 setting, resulting in improved runtime performance.

To demonstrate our approach, we build a LatticeBox runtime that runs WebAssembly-WASI applications within isolated compartments. The runtime is based on two components: wasm2c, which compiles Wasm bytecode into C source code, and uvwasi, which provides system call support for command-line arguments, I/O, etc. Importantly, integrating LatticeBox support into the existing runtime requires minimal changes. Wasm already provides a structured and sandboxed execution model, with a well-defined separation between code and data, and explicit function entry points. These properties align well with LatticeBox's label-based isolation.

1) LatticeBox Runtime Initialization: Before launching a Wasm application, the runtime sets up an isolated execution environment. It maps the compiled Wasm code with a non-zero lattice label, allocates a fresh stack, and labels both the stack and linear memory with the same lattice label. It then saves its own register state—including ra (return address) and sp (stack pointer)—and sets the return address to a designated exit entry point in the trusted root compartment. This ensures a secure transition back after the application finishes.

2) Service Function Invocation: For system services such as memory growth or I/O (via uvwasi), we introduce wrapper functions that act as trusted gateways. Each wrapper begins with an `lp_land` instruction to escalate permissions and switch to the trusted stack. The wrapper then invokes the actual service function and, upon return, restores the original stack before transferring control back to the Wasm code.

Thanks to LatticeBox's hardware support for return address integrity, the ra register at the return site can be trusted, ensuring control returns only to the legitimate call site. Additionally, because these service functions do not support callbacks, they do not need to preserve callee-saved registers, which simplifies their implementation and reduces overhead.

3) libc Integration: We also make minimal changes to libc by inserting an `lp_land` instruction before functions that are directly invoked by wasm2c-generated code. These include floating-point operations and memory utility functions. Although no permission escalation occurs in these cases, the `lp_land` instruction still serves as a controlled entry point into shared runtime code, ensuring consistent enforcement of LatticeBox's execution model.

4) Application Termination: When the Wasm application completes, it must call the exit entry in the trusted root compartment. This exit function restores the original register state, tears down the Wasm application's resources, and securely dismantles its lattice box.

5) Containment and Security Assurance: Even if an attacker gains full control over the Wasm application—for example, by exploiting a vulnerability in the compiler or injecting malicious bytecode—they remain confined within the lattice box. They cannot access memory outside of the compartment, since all memory loads and stores are hardware-checked against the current lattice label. They also cannot call privileged functions directly, as any attempt to jump across compartments without going through a valid `lp_land` entry will either lower the execution label or raise an exception. Return address forgery is prevented through hardware enforcement, and since the application cannot change its own label or map memory, privilege escalation is fundamentally blocked.

B. Kernel Module Isolation

In Linux, dynamically loadable kernel modules extend the base kernel's functionality after boot, providing support for device drivers, file systems, and other components. While modular and pluggable by design, these modules still execute with full kernel privileges once loaded. This broad access far exceeds what most modules require for their functionality and introduces substantial risk. Notably, drivers are responsible for a significant portion of kernel vulnerabilities. Microkernel architectures have shown that much of this code can operate safely with reduced privileges.

To address this over-privileging, we propose isolating kernel modules using lattice boxes. Our approach confines modules within restricted permission domains enforced at the hardware level. As a proof of concept, we implement a lattice-boxed version of the IPv6 kernel module. More generally, LatticeBox supports isolating any kernel code that does not directly manage memory or processor state, making this approach widely applicable within the kernel.

The implementation is conceptually similar to the user-mode WebAssembly runtime described earlier. The module's code is mapped with a non-zero (low-privilege) label, and it runs on a private stack. Trampolines are used to switch stacks during control transfers. However, the kernel context introduces two additional challenges:

• Interleaved Call Stack: The call stack interaction between the IPv6 module and base kernel is more complex than user-space wasm runtimes. Many network-related functions are shared between IPv4 and IPv6, leading to frequent transitions between base kernel and IPv6-specific code. For example, IPv6 code invokes helper routines in the kernel, and kernel functions later invoke callbacks into the IPv6 module. To preserve correctness and security during such transitions, any base kernel function calling into the IPv6 module must save callee-saved registers before the call and restore them on return.

• Buffer Sharing: The IPv6 module occasionally accesses memory buffers allocated by the base kernel. Ideally, we would identify which of these allocations are later used by the module and allocate them from a dedicated low-security-level heap. However, performing such identification requires non-trivial pointer analysis, which is complex and outside the scope of this work. As a prototype, we forgo fine-grained heap isolation and instead map the entire kernel heap with a low-permission label, allowing the IPv6 module to access shared buffers safely. We leave automated pointer analysis and heap isolation to future work.

C. Compatibility and Porting Effort

The hardware design is fully forward-compatible, allowing unmodified operating systems and software to run on a LatticeBox-enabled processor without changes or performance overhead. Enabling LatticeBox functionality requires minimal

effort. Porting the kernel takes fewer than 500 lines of code, covering page table management and context-switching modifications. Porting a wasm2c runtime requires under 1,000 lines of changes, including adding a reusable lattice box runtime library and using macros to annotate and wrap service functions called by WebAssembly code.

VII. Evaluation

A. Security Analysis

1) Permission Monotonicity: LatticeBox enforces permission monotonicity, meaning no compartment can escalate its permission or create a new one with a higher level, even with arbitrary code execution. This is ensured by restricting two key capabilities to the most privileged level (the lattice upper bound): executing sensitive instructions and modifying PTEs. Without syscall or page table modifications, lattice-boxed code cannot escalate its permission in our threat model. To perform sensitive operations that could break permission monotonicity, such as modifying or creating page table entries, the code must call wrapper functions in the TCB. The lp_land instruction records the caller's permission level, and the remaining enforcement is handled by software.

2) Cross-Domain Control-Flow Integrity: Another potential attack vector involves sandboxed code tricking a trusted compartment into following an incorrect control flow to perform operations on its behalf. To prevent this, LatticeBox enforces cross-domain control-flow integrity. This has two key aspects. First, a lower-privileged compartment can only jump to designated service function entry points, not internal labels or mid-function addresses. Hardware ensures that all cross-permission jumps target an lp_land instruction. Second, when a privileged compartment finishes a service function, it must securely return to the caller to prevent Return-Oriented Programming (ROP) attacks. Hardware enforces this by requiring the instruction before lp_land to write to a return address (RA) register, preventing return path forgery. The system also tracks the last committed instruction's permission level and return address to restore it after interruptions. Regarding callbacks, the TCB does not support arbitrary callbacks from untrusted domains. If needed, callbacks must go through an explicit indirection that transitions to a lower-privileged compartment, preventing control-flow hijacks into privileged code.

3) Multi-Core Synchronization: Updating a Lattice Permission Key requires modifying the PTEs, which triggers a TLB flush to prevent cores from using stale LPKs, similar to updating basic read, write, or execute permissions. Although TLB flushes are costly, their frequency is minimized by design, as memory regions with distinct labels are long-lived, keeping the performance overhead low.

4) Spectre: LatticeBox is invulnerable to Spectre-class attacks discussed in HFI [31] and SpecMPK [32] because it enforces correct memory permissions even during transient execution. Permissions are bound to each instruction at issue time and then validated at the point of memory access. Within the LatticeBox, malicious code cannot access higher-level memory. Even if a cross-compartment jump is speculatively executed before a memory access, the memory instruction still enforces its original permissions. This design also defends against attacks where the attacker tricks the trusted runtime into speculatively executing malicious code. The moment the runtime jumps to malicious code—speculatively or not—its permissions are immediately lowered.

B. Experimental Setup

We instantiated a BOOM core and additional system-on-chip (SoC) components using the Chipyard [33] framework. Due to FPGA resource constraints and to reflect realistic yet practical hardware configurations, we selected the SmallConfig preset, which retains key out-of-order execution features with manageable resource overhead. Unused interfaces and the L2 cache were removed to further reduce resource usage and ensure stable operation at 100 MHz.

For LatticeBox, we allocated 10-bit LPKs to encode permission levels. The system runs Linux kernel version 6.1, compiled with LLVM 15.0.7. User-space binaries were compiled with the same compiler and linked against LLVM runtime components—compiler-rt, libc++, libc++abi, and libunwind—along with musl libc version 1.2.5. The target FPGA platform is a Xilinx Kintex UltraScale (XCKU060) development board with 4 GB of onboard RAM.

C. Hardware Resource Usage

Table I summarizes LatticeBox's FPGA resource overhead. Lookup tables (LUTs) increased by 0.68%, and flip-flops (FFs) by 3.03%, which are comparable to overheads reported in prior work. For instance, Donky, which extends Intel MPK, reports +1.8% LUT and +0.9% FF increases. XPC, a design aimed at accelerating inter-process communication (IPC), reports a 2.0% increase in LUTs. SecureCells reports a reduction in FPGA resource usage; however, this is due to its removal of the paging subsystem, which renders it incompatible with virtual memory and existing software stacks. Additionally, unlike these prior works, which are based on in-order processor cores, our design uses a realistic out-of-order BOOM core. This more closely reflects industrial-grade architectures and highlights the practical viability of LatticeBox.

Finally, LatticeBox has minimal impact on timing: both the baseline and LatticeBox-enhanced BOOM designs meet timing closure at 100 MHz. The worst negative slack in both cases was observed in the branch predictor module, which remains unmodified in our implementation.

D. System Performance

1) Microbench: To evaluate the efficiency of LatticeBox during compartment transitions, we conducted a series of microbenchmark experiments and compared the results with prior work, as shown in Table II. We measured the latency introduced in four scenarios: (1) permission switching only, (2) permission switching with stack switching, (3) full context switching, and (4) full context switching with register clearing. Each scenario was executed in a loop 1,000 times, and the

TABLE I: Hardware resource cost of the baseline and LatticeBox when synthesized on an FPGA.

	RISC-V Boom Cores					Whole Systems					
	#LUT	%	#FF	%	#BRAM	#LUT	%	#FF	%	#BRAM	Worst Neg Slack (ns)
baseline	75,968	_	47,325	_	11	99,454	_	81,059	_	106	0.021
LatticeBox	76,483	+0.68%	48,759	+3.03%	11	99,974	+0.52%	82,520	+1.80%	106	0.001

average of the last 500 iterations was used as the final result to minimize warm-up effects and measurement noise.

We report round-trip latency for all transitions. For prior work that only reports one-way transition latency, we double the published value to estimate a round-trip comparison. As shown in Table II, LatticeBox achieves extremely low switching latency. In the simplest case where only permission switching is required, the overhead is just 1 cycle. This significantly outperforms Intel MPK-based designs such as ERIM (approximately 99 cycles) and BULKHEAD (approximately 408 cycles). Even in the most demanding scenario, which includes full context switching and register clearing, the round-trip latency is only 94 cycles. This performance surpasses prior Intel MPK-based solutions, which incur higher cycle counts for basic permission updates. While XPC and SecureCells achieve competitive latency, LatticeBox maintains an advantage through its non-blocking pipeline architecture that eliminates stalling during permission transitions.

2) Macrobench: We evaluate the performance impact of integrating LatticeBox into the system using the SPEC CPU2006 benchmark suite. Although SPEC CPU2017 is more modern, it demands 16GB of memory, which exceeds the capacity of our FPGA development board. Therefore, we use CPU2006 instead. In the SPEC CINT2006 subset, benchmark 400.perlbench is excluded due to compilation issues. All benchmarks are compiled with -O2 optimization and statically linked. To eliminate I/O effects, all executables and input data are loaded onto a tmpfs (in-memory file system) prior to execution.

Each test program is run on two configurations: the baseline system (unmodified BOOM and Linux) and the LatticeBox-enhanced system. The enhanced configuration includes hardware support for label enforcement and kernel modifications for managing execution labels and memory mapping with lattice labels. However, for this evaluation, no actual lattice boxes are instantiated; we measure only the baseline overhead introduced by the underlying support.

Performance is measured using BOOM's performance counters, specifically by recording the total number of cycles per benchmark. All benchmarks complete successfully, demonstrating that LatticeBox maintains full backward compatibility. For all tested workloads, the slowdown is less than 0.2%, indicating that LatticeBox introduces negligible performance overhead for systems not actively using its security features.

E. Wasm

For the WebAssembly (Wasm) runtime enhancement evaluation, we also use the SPEC CPU2006 benchmark suite. However, Wasm remains significantly less performant than native execution. Running the full reference workload under the baseline WebAssembly configuration would require approximately three weeks. To accelerate the evaluation while preserving representative behavior, we instead use the smaller train workload. Benchmarks are executed under three configurations: (1) baseline WebAssembly, (2) WebAssembly with bounds-checking instrumentation, and (3) WebAssembly protected by LatticeBox. Figure 4 presents normalized results for configurations (2) and (3) relative to the baseline. The LatticeBox-enhanced Wasm runtime successfully completes all test cases, demonstrating that our modifications maintain full execution compatibility. Performance analysis reveals consistently low overhead, with all test cases showing less than 2% slowdown and a geometric mean of just 0.3%, significantly outperforming the bounds-checking instrumentation approach, which incurs near 20% overhead on average. One benchmark, 464.h264, shows unexpected performance improvement, which we tentatively attribute to cache line alignment effects introduced by our code modifications.

The performance cost of LatticeBox primarily comes from the added trampolines invoked during service calls made by Wasm applications. Each trampoline introduces one `lp_land` instruction, two stack switches, and a function call round-trip. As shown in our microbenchmark results, these operations take fewer than 100 cycles. Since service calls are relatively infrequent, the overall performance impact remains negligible. Furthermore, memory access checks within lattice boxes execute entirely in hardware, eliminating instrumentation overhead. This combination of rare trampoline costs and hardware-accelerated memory protection makes LatticeBox both performant and practical for building compartmentalized systems. The binary size overhead for all WASM executables targeting LatticeBox is under 1%, significantly less than the 5-10% overhead from bounds-checking instrumentation.

F. Apache Bench

We also evaluate the performance impact of isolating the ipv6 kernel modules with LatticeBox using ApacheBench. Specifically, we use ApacheBench to retrieve files of sizes 100KB, 1MB, and 10MB. Each file is requested 1,000 times, and the data transfer rate is recorded. To reduce variability, each experiment is repeated 10 times and averaged. We test with (1) the base kernel with IPv6 support, and (2) the LatticeBox support kernel with the IPv6 module isolated into a lattice box. The results of isolated module normalized to the base kernel are shown in Figure 5, along with comparisons to BULKHEAD and HAKC.

LatticeBox achieves 97.1%, 97.3%, and 97.7% of baseline performance on file sizes of 100KB, 1MB, and 10MB, respectively. These results are comparable to BULKHEAD and

TABLE II: The micro-benchmark test results of LatticeBox and comparison with related works.

	Latency in Cycles (Round Trip)				Platform		
	Permission Switch	Stack Switch	Full Context Switch	Register Clear	Processor	OS	Method
ERIM	99	/	/	/	Xeon Gold 6142	Linux v4.9	Fabricated
Donky	/	2136	/	/	Rocket	Linux v5.1	FPGA
Donky	/	428	/	/	Xeon 8275CL	Linux v5.3	Fabricated
XPC	41	66*	162*	/	Rocket	seL4	FPGA
SecureCells	16	/	/	/	Rocket	seL4	QEMU/FPGA
BULKHEAD	408*	447*	/	/	Intel Core i7-12700H	Linux v6.1	Fabricated
LatticeBox	1	39	82	94	BOOM	Linux v6.1	FPGA

*: Obtained via double the provided single trip latency

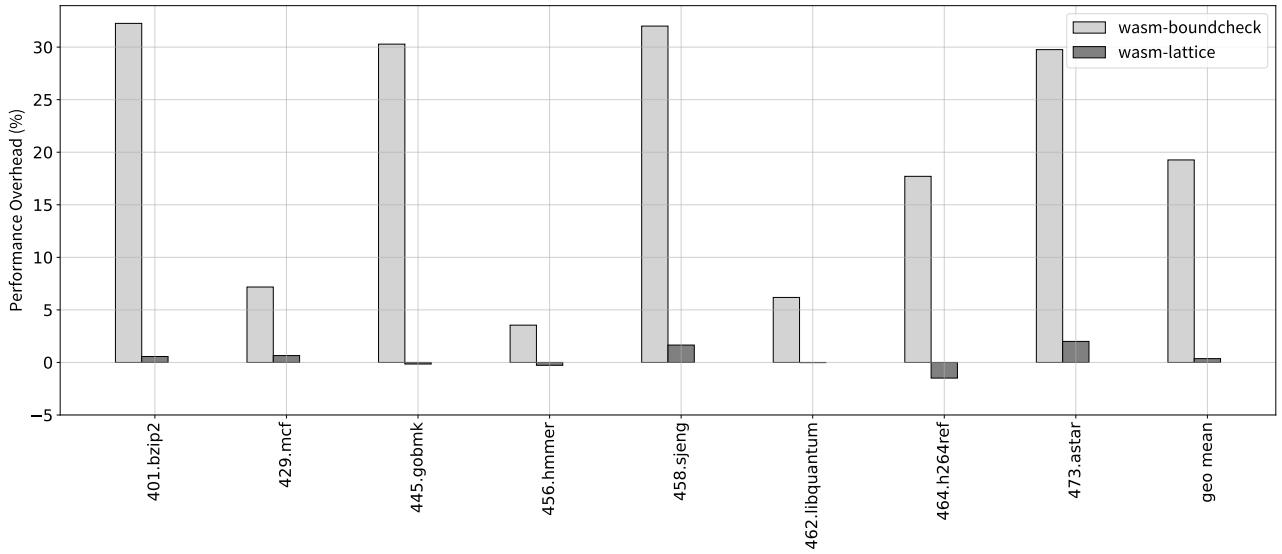


Fig. 4: Relative runtime overhead of LatticeBox on SPEC CINT2006

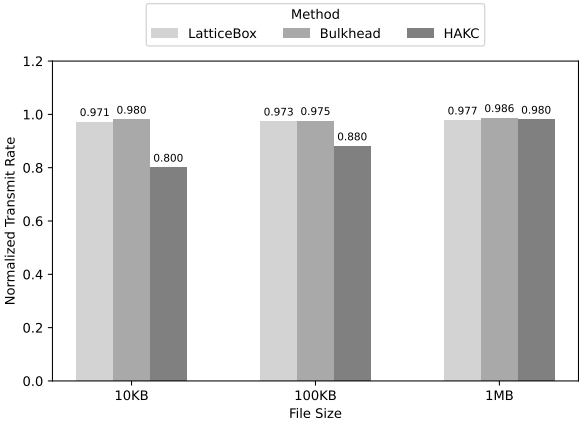


Fig. 5: Normalized throughput of isolated IPv6 module (via LatticeBox), compared with BULKHEAD and HAKC.

significantly outperform HAKC. In theory, LatticeBox should provide even better performance than BULKHEAD due to its lower-latency domain transitions and avoidance of page faults during memory sharing. The slight performance gap may stem from platform differences or implementation details.

Regardless, LatticeBox offers significantly stronger security guarantees. BULKHEAD focuses primarily on preventing simple memory read/write violations and assumes that attackers cannot control registers or execute arbitrary code. In contrast, LatticeBox adopts a much stronger threat model, defending even against attackers who achieve arbitrary code execution within an isolated kernel module. It enforces strict hardware-level boundaries, ensuring that critical parts of the kernel remain protected regardless of compromises within isolated components.

VIII. Related Work

A. Software Fault Isolation

Software Fault Isolation (SFI) is a software-based isolation technique initially proposed by Wahbe et al. [11]. It inserts dynamic safety checks to prevent untrusted code from accessing memory beyond a designated sandbox region. While this approach avoids the need for hardware modifications or specialised OS support, it suffers from several critical limitations. First, the injected checks slow programs; even highly optimized deployments such as Google Native Client report slowdowns between 5-20% on SPEC workloads [34]. Second, most SFI implementations rely on masking each address with a

fixed base and size [12], [35], which confines every sandbox to a power-of-two memory region. This constraint wastes address space and is especially restrictive on 32-bit systems. Third, SFI guards are not fool-proof. Real-world exploits against Native Client and the Chrome V8 sandbox have used corner-case instruction sequences and compiler omissions to bypass the checks and escape the sandbox [36], [37].

B. Intel MPK Improvements

As discussed, Intel Memory Protection Keys (MPK) face inherent limitations in scalability, security, and performance. Prior work has attempted to address these issues through software and hardware extensions. To overcome MPK's 4-key hardware limit, systems such as libmpk [13] and VDom [16] introduce a virtual key mechanism to map more memory domains in software. EPK [15] and BULKHEAD [18] use paging as the second isolation layer. Regarding security, ERIM [38] mitigates the requirement of CFI by introducing call gates to enforce entry points, using the `WRPKRU`/`RDPKRU` instruction pair—which validate permission transitions at runtime. This design has been widely adopted in subsequent work.

While these approaches improve scalability or security, they inevitably introduce more overheads, degrading performance. Fundamental optimizations remain impossible without hardware modifications. Donky [17] proposes a hardware redesign, extending the key bitwidth to 10 bits. However, it can hold at most 4 keys at any given time. When the accessed key is not loaded, the system needs to trap and fetch the required key value from protected memory, introducing a significant miss penalty. Although Donky introduces a monitor state to manage PKRU updates, which achieves good security. But it comes at a cost, trap is less performant than jump as it is hard to predict in execution, hampering high-performance processors.

C. Isolation based on other Existing Hardware Primitives

Several recent systems retrofit existing hardware security primitives to enforce intra-process isolation. SEIMI [39], PANIC [40], and CETIS [41] repurpose widely available CPU features such as Supervisor Mode Access Prevention (SMAP), Privileged Access Never (PAN), and Intel Control-flow Enforcement Technology (CET). Although these solutions achieve low runtime overhead by cleverly repurposing existing hardware features, their effectiveness is inherently limited by the granularity and semantics of the original mechanisms; thus, they cannot support arbitrary isolation policies or provide comprehensive memory protection without additional performance or complexity trade-offs.

Other work has explored combining existing hardware primitives with software-level techniques to strengthen isolation guarantees. HIVE [42] employs ARM Pointer Authentication (PA) instructions to sandbox eBPF programs within the Linux kernel, reducing privilege by embedding authentication codes directly into kernel pointers. HAKCs [29] implement a two-layer isolation scheme by encoding capability-like tokens into pointer metadata, using these tokens to validate each memory access efficiently. While these approaches improve isolation without requiring dedicated hardware modifications, they introduce overhead proportional to the frequency of pointer validations. In contrast, LatticeBox significantly reduces runtime overhead while maintaining strong security guarantees, overcoming the granularity and efficiency limitations inherent in prior hardware-primitive-based isolation solutions.

D. Hardware-Software Co-design Approaches

Recent isolation frameworks are increasingly leveraging hardware-software co-design to achieve security and performance. CODOMs [43] introduces hardware-supported, code-centric memory domains that bind permissions directly to code segments rather than processes. Each domain is associated with code boundaries, with permissions being enforced by custom hardware logic to ensure that memory operations can only access permitted regions. Similarly, SecureCells [30] proposes a compartmentalised architecture that uses hardware extensions to enforce strong isolation between compartments at memory granularity. SecureCells achieves this by associating separate hardware-enforced permission tables with each compartment, effectively preventing unauthorised cross-compartment access.

Another notable example of hardware-software co-design is XPC [44], which specifically targets efficient and secure inter-process communication (IPC). XPC introduces dedicated hardware support that facilitates rapid context switching and zero-copy message passing between isolated processes without kernel intervention. At its core, XPC introduces "relay-seg", which are special memory segments that are mapped directly to physical memory for inter-process message transfers. This design substantially reduces the overhead associated with traditional IPC mechanisms involving kernel mediation and data copying, achieving significant performance improvements for cross-domain communication.

Although these co-designed hardware-software systems significantly advance isolation technology, each approach introduces non-trivial hardware modifications and increased complexity. CODOMs and SecureCells require customised permission-checking hardware and extended memory management structures. Similarly, XPC relies on specialised memory mappings and additional hardware logic to manage message passing and context switches. LatticeBox also follows a hardware-software co-design philosophy, but its footprint is deliberately modest. It minimises architectural complexity and hardware changes by leveraging compact, lattice-based permission labels that are embedded directly within standard page-table entries. This enables LatticeBox to provide scalable, low-overhead isolation while striking a balance between hardware simplicity and strong security guarantees.

E. Isolation Beyond Memory

Recent studies have explored isolation mechanisms that extend beyond traditional memory protection. While μSwitch [45] and ISA-grid [46] target resource-level and instruction-level isolation, respectively, LatticeBox offers a general, scalable memory-isolation framework that could be

integrated with such complementary isolation solutions to provide holistic security guarantees across multiple system dimensions.

IX. Conclusion

We have presented LatticeBox, a hardware-software co-designed framework that introduces lattice-based access control into memory protection to address the limitations of existing isolation mechanisms. By representing access permissions as hierarchical N-bit vectors and embedding them into page table entries, LatticeBox enables fine-grained, scalable, and secure compartmentalization with single-cycle privilege transitions and minimal hardware overhead. Our RISC-V implementation and evaluation on real-world workloads demonstrate that LatticeBox achieves domain switching up to 180× faster than Intel MPK, while supporting significantly more isolation domains. LatticeBox offers a practical and robust foundation for next-generation isolation in browsers, operating systems, and secure runtimes.

Acknowledgment

We would like to thank the shepherd and anonymous reviewers for their insightful comments and suggestions that greatly improved the quality of this paper. This work was supported by the National Natural Science Foundation of China (No.62502468, No.U24A20337), the Zhongguancun Laboratory and the Joint Research Center for System Security (JCSS), Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

References

[1] A. Barth, C. Jackson, C. Reis, and the Google Chrome Team, "The security architecture of the chromium browser," Dec. 2008. [Online]. Available: https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf

[2] "Security/Sandbox – MozillaWiki." [Online]. Available: https://wiki.mozilla.org/Security/Sandbox

[3] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the firefox renderer," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 699–716. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[4] J. Liedtke, "On micro-kernel construction," SIGOPS Oper. Syst. Rev., vol. 29, no. 5, p. 237–250, Dec. 1995. [Online]. Available: https://doi.org/10.1145/224057.224075

[5] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[6] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," ACM Comput. Surv., vol. 53, no. 1, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3371038

[7] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[8] Y. Li, A. Bhattacharyya, M. Kumar, A. Bhattacharjee, Y. Etsion, B. Falsafi, S. Kashyap, and M. Payer, "Single-address-space faas with jord," in Proceedings of the 52nd Annual International Symposium on Computer Architecture, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 694–707. [Online]. Available: https://doi.org/10.1145/3695053.3731108

[9] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, 2024, volume 3A, Section 3.5: 64-Bit Mode. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html

[10] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and vm functions," in Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments (VEE '20), 2020, pp. 157–171, table 1 shows page-table-switch IPC costs 834 cycles even with tagged TLBs. [Online]. Available: https://arkivm.github.io/publications/2020-vee-lvds.pdf

[11] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," SIGOPS Oper. Syst. Rev., vol. 27, no. 5, p. 203–216, Dec. 1993. [Online]. Available: https://doi.org/10.1145/173668.168635

[12] Z. Yedidia, "Lightweight fault isolation: Practical, efficient, and secure software sandboxing," in Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 649–665. [Online]. Available: https://doi.org/10.1145/3620665.3640408

[13] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel MPK)," in 2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, Jul. 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon

[14] E. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[15] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in 2022 USENIX Annual Technical Conference (USENIX ATC 22). Carlsbad, CA: USENIX Association, Jul. 2022, pp. 609–624. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/gu-jinyu

[16] Z. Yuan, S. Hong, R. Chang, Y. Zhou, W. Shen, and K. Ren, "Vdom: Fast and unlimited virtual domains on multiple architectures," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 905–919. [Online]. Available: https://doi.org/10.1145/3575693.3575735

[17] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient In-Process isolation for RISC-V and x86," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[18] Y. Guo, Z. Wang, W. Bai, Q. Zeng, and K. Lu, "BULKHEAD: secure, scalable, and efficient kernel compartmentalization with PKS," in 32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025. The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/bulkhead-secure-scalable-and-efficient-kernel-compartmentalization-with-pks/

[19] saelo@, "V8 Sandbox - Hardware Support," https://docs.google.com/document/d/12MsaG6BYRB-jQWNkZiuM3bY8X2B2cAsCMLLdgErvK4c/, February 2024.

[20] R. Sandhu, "Lattice-based access control models," Computer, vol. 26, no. 11, pp. 9–19, 1993.

[21] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," May 2020.

[22] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," SIGPLAN Not., vol. 52, no. 6, p. 185–200, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[23] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 217–234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[24] "WebAssembly Core Specification." [Online]. Available: https://www.w3.org/TR/wasm-core-2/

[25] A. S. Tanenbaum and H. Bos, Modern Operating Systems, 4th ed. Pearson, 2015, chapter 2 "Operating-System Structures" discusses monolithic and microkernel designs.

[26] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," ACM Transactions on Computer Systems, vol. 34, no. 1, pp. 1:1–1:29, 2016, §4 compares IPC latency to Linux syscalls in high-throughput workloads.

[27] R. S. Sandhu, "Lattice-based access control models," IEEE Computer, vol. 26, no. 11, pp. 9–19, Nov. 1993.

[28] D. E. Denning, "A Lattice Model of Secure Information Flow," in Proceedings of the Fifth ACM Symposium on Operating Systems Principles, Nov. 1976, pp. 236–243. [Online]. Available: http://faculty.nps.edu/dedennin/publications/lattice76.pdf

[29] D. P. McKee, Y. Giannaris, C. Ortega, H. E. Shrobe, M. Payer, H. Okhravi, and N. Burow, "Preventing kernel hacks with hakcs." in NDSS. The Internet Society, 2022.

[30] A. Bhattacharyya, F. Hofhammer, Y. Li, S. Gupta, A. Sanchez, B. Falsafi, and M. Payer, "Securecells: A secure compartmentalized architecture," in 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 2921–2939.

[31] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 266–281. [Online]. Available: https://doi.org/10.1145/3582016.3582023

[32] D. Adak, H. Zhou, E. Rotenberg, and A. Awad, "Specmpk: Efficient in-process isolation with speculative and secure permission update instruction," in 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2025, pp. 394–408.

[33] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," IEEE Micro, vol. 40, no. 4, pp. 10–21, 2020.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in 30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA. IEEE Computer Society, 2009, pp. 79–93. [Online]. Available: https://doi.org/10.1109/SP.2009.25

[35] V8 Developers. (2024, Apr.) The V8 Sandbox. [Online]. Available: https://v8.dev/blog/sandbox

[36] G. P. Zero, "CVE-2021-30551: V8 sandbox escape," https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30551.html, 2021, accessed 3 Aug. 2025.

[37] "cve-2011-3020," https://nvd.nist.gov/vuln/detail/cve-2011-3020, 2011.

[38] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[39] Z. Wang, C. Wu, M. Xie, Y. Zhang, K. Lu, X. Zhang, Y. Lai, Y. Kang, and M. Yang, "Seimi: Efficient and secure smap-enabled intra-process memory isolation," in 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 592–607.

[40] J. Xu, M. Xie, C. Wu, Y. Zhang, Q. Li, X. Huang, Y. Lai, Y. Kang, W. Wang, Q. Wei, and Z. Wang, "Panic: Pan-assisted intra-process memory isolation on arm," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 919–933. [Online]. Available: https://doi.org/10.1145/3576915.3623206

[41] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2989–3002. [Online]. Available: https://doi.org/10.1145/3548606.3559344

[42] P. Zhang, C. Wu, X. Meng, Y. Zhang, M. Peng, S. Zhang, B. Hu, M. Xie, Y. Lai, Y. Kang, and Z. Wang, "HIVE: A hardware-assisted isolated execution environment for eBPF on AArch64," in 33rd USENIX Security Symposium (USENIX Security 24). Philadelphia, PA: USENIX Association, Aug. 2024, pp. 163–180. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-peihua

[43] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "Codoms: protecting software with code-centric memory domains," in Proceeding of the 41st Annual International Symposium on Computer Architecture, ser. ISCA '14. IEEE Press, 2014, p. 469–480.

[44] D. Du, Z. Hua, Y. Xia, B. Zang, and H. Chen, "Xpc: Architectural support for secure and efficient cross process call," in 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), 2019, pp. 671–684.

[45] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij, "μSwitch: Fast Kernel Context Isolation with Implicit Context Switches," in 2023 IEEE Symposium on Security and Privacy (SP), 2023, pp. 2956–2973.

[46] S. Fan, Z. Hua, Y. Xia, H. Chen, and B. Zang, "ISA-Grid: Architecture of Fine-grained Privilege Control for Instructions and Registers," in Proceedings of the 50th Annual International Symposium on Computer Architecture, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589050