ISA-Grid: Architecture of Fine-grained Privilege Control for Instructions and Registers

Shulin Fan
forestree@sjtu.edu.cn
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems(Ministry of Education)
Shanghai, China

Zhichao Hua*
zchua@sjtu.edu.cn
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems(Ministry of Education)
Shanghai, China

Yubin Xia
xiayubin@sjtu.edu.cn
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai AI Laboratory
Engineering Research Center for Domain-specific Operating Systems(Ministry of Education)
Shanghai, China

Haibo Chen
haibochen@sjtu.edu.cn
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems(Ministry of Education)
Shanghai, China

Binyu Zang
byzang@sjtu.edu.cn
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems(Ministry of Education)
Shanghai, China

ABSTRACT

Isolation is a critical mechanism for enhancing the security of computer systems. By controlling the access privileges of software and hardware resources, isolation mechanisms can decouple software into multiple isolated components and enforce the principle of least privilege. While existing isolation systems primarily focus on memory isolation, they overlook the isolation of instruction and register resources, which we refer to as ISA (Instruction Set Architecture) resources. However, previous works have shown that exploiting ISA resources can lead to serious security problems, such as breaking the system's memory isolation property by abusing x86's *CR3* register. Furthermore, existing hardware only provides privilege-level-based access control for ISA resources, which is too coarse-grained for software decoupling. For example, ARM Cortex A53 has several hundred system instructions/registers, but only four exception levels (EL0 to EL3) are provided. Additionally, more than 100 instructions/registers for system control are available in only EL1 (the kernel mode). To address this problem, this paper proposes ISA-Grid, an architecture of fine-grained privilege control for instructions and registers. ISA-Grid is a hardware extension that enables the creation of multiple ISA domains, with each domain having different privileges to access instructions and registers. The ISA domain can provide bit-level fine-grained privilege control for registers. We implemented prototypes of ISA-Grid based on two different CPU cores: 1) a RISC-V CPU core on an FPGA board and 2) an x86 CPU core on a simulator. We applied ISA-Grid to different cases, including Linux kernel decomposition and enhancing existing security systems, to demonstrate how ISA-Grid can isolate ISA resources and mitigate attacks based on abusing them. The performance evaluation results on both x86 and RISC-V platforms with real-world applications showed that ISA-Grid has negligible runtime overhead (less than 1%).

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Security and privacy** → **Systems security**.

KEYWORDS

Privilege Control, Instruction Set Architecture, Software Isolation

ACM Reference Format:
Shulin Fan, Zhichao Hua, Yubin Xia, Haibo Chen, and Binyu Zang. 2023. ISA-Grid: Architecture of Fine-grained Privilege Control for Instructions and Registers. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589050

*Corresponding author: Zhichao Hua

1 INTRODUCTION

Isolation is a crucial mechanism for improving the security of both user applications [14, 30, 42, 52, 59, 67] and system software [21, 31, 44, 49, 50, 78]. By breaking software down into multiple isolated components, isolation makes vulnerabilities in one component have less impact on the others. Each component is assigned limited privileges, in accordance with the principle of least privilege, to enhance software security [34].

Isolation mechanisms aim to create multiple domains and limit their access to specific hardware and software resources. There has been a long line of research on building isolation mechanisms based on various hardware or software platforms by designing new system-level architectures [21, 30, 31, 43, 78], using compiler methods [14, 17, 20, 25, 45], leveraging existing hardwares [27, 29, 49, 52, 57, 61, 62], or adding new hardware features [59, 65, 72, 73]. However, the majority of these works concentrate exclusively on isolating memory, which is just one type of system resource. As a result, they neglect the isolation of instructions and registers, which we refer to as ISA (Instruction Set Architecture) resources.

Abusing Instruction Set Architecture (ISA) resources can result in serious security problems in both hardware and software systems [11, 15, 19, 36, 48, 51, 54, 64, 77, 79]. For instance, using cycle registers, such as the *rdtsc* of x86, can increase the success rate of timing-based side-channel attacks [77], while manipulating CPU frequency and voltage control registers, such as *MSR 0x150* in x86, can facilitate voltage-based attacks [36, 48, 54]. Moreover, the abuse of certain system control registers, such as the *CR3* in x86, can compromise the entire system's security property. Many attacks are based on the assumption that the attacker has the ability to use specific instructions/registers. For example, the Stealthy Page Table-Based Attack [64] assumes that the attacker can set the *CD* bit of x86's *CR0*, while the Controlled-Channel Attack [77] assumes that the attacker can replace the fault handler by modifying *IDTR* or changing IDT. In the V0LTpwn Attack [36], the attacker must have access to *MSR 0x150.* We call such attacks whose prerequisite is access to certain ISA resources *ISA-abuse-based attacks.*

On the contrary, unlike memory, existing hardware only provides privilege-level-based access control for ISA resources, which is too coarse-grained for decoupling software. All software components at the same exception level, such as the Linux kernel and all system-level services in the kernel space, share the same privilege on ISA resources. Although existing methods attempt to perform fine-grained privilege control for ISA resources by scanning the binary and replacing all possible binary code that may access illegal instructions/registers [10, 21, 27, 32, 76], such methods significantly increase development effort and lack proof of correctness and completeness. Currently, such a method is only applicable to specific instructions or registers and is impractical for generic access control for ISA resources, particularly for ISAs with variable-length instructions like x86 [56].

To solve the above problem, this paper introduces ISA-Grid, an architecture of fine-grained privilege control for instructions and registers. ISA-Grid is a hardware extension that enables software to create multiple ISA domains, with each domain having different privileges to access ISA resources. Our system can provide bit-level fine-grained privilege control for registers that contain multiple function fields. The design of the ISA-Grid includes three parts. First, ISA-Grid introduces a **hybrid privilege structure** to perform the privilege control with different granularities for different instructions and registers. Second, an **unforgeable domain switching method** is provided for the secure switch between different ISA domains. Finally, ISA-Grid uses a **domain privilege cache** to speed up the privilege check procedure.

To demonstrate the effectiveness of ISA-Grid, we implemented the proposed hardware extension in a RISC-V Rocket Core [9] on an FPGA board and an x86 core on a cycle-accurate simulator. We then applied ISA-Grid to different use cases to showcase its mitigation capabilities. Firstly, we used ISA-Grid to decompose the Linux kernel, where most of the Linux code runs in a de-privileged domain that can only access general computing instructions/registers and read a few other registers. Privileges of special instructions/registers are only granted to the code that requires them. Secondly, we employed ISA-Grid to construct a nested monitor in the Linux kernel for memory protection. We performed a detailed evaluation using the x86 and RISC-V prototypes of ISA-Grid. The evaluation results demonstrate that ISA-Grid provides the current system with the capability to control instruction/register privileges at a fine granularity while maintaining a very limited runtime overhead.

This work makes the following contributions:

- We analyze the importance of privilege control of ISA resources, which previous works overlook.
- We provide the design of ISA-Grid, a new hardware architecture of fine-grained privilege control for instructions and registers. The design of ISA-Grid does not bind to a specific platform and can extend to different CPU implementations.
- We implement ISA-Grid on both RISC-V and x86 platforms and use it in several use cases to show how it can help to improve system security.

2 BACKGROUND AND MOTIVATION

2.1 The Complex ISA Resources

The ISA resources, which include instructions and registers, are the crucial resource provided by the CPU. However, modern commercial processors have intricate ISAs, with hundreds of instructions and registers for system management, as seen in the ARM Cortex A53 [6]. For instance, the EL1 mode contains over 100 system control instructions and registers, such as identification registers, exception handling registers, virtual memory control registers, and TLB maintaining instructions. Similarly, x86 processors have even more hardware features, leading to more instructions and registers. Notably, every instruction or register may be complex to use. Taking the 32-bit *SCTLR_EL1* register on ARM as an example, it can have up to 28 different function fields, illustrating the intricacy of these resources. Moreover, with CPU vendors continually adding new hardware features, the ISA resources of commercial processors are rapidly expanding. For example, ARMv6 has only 30+ system registers [7], while ARMv9 grows to 400+ [5].

2.2 The Need for ISA Resources Access Control

Pure memory isolation, in other words, memory access control, cannot de-privilege a piece of code or selectively grant code privileges to execute certain instructions. However, access control of ISA resources is necessary for software isolation, which can complete the principle of least privilege.

Table 1: Some of ISA-abuse-based attacks.

Attack	Architecture	Reg./Inst. as Prerequisites	Consequence	Can ISA-Grid mitigate
Controlled-Channel Attacks [77]	x86	IDTR	Stealing data from different types of TEEs.	✓
FORESHADOW Attacks [63]	x86	*wbinvd* instruction, DR0-7	Extracting enclave secrets.	✓
NAILGUN Attacks [51]	ARM	PMU registers	Stealing sensitive data.	✓
Stealthy Page Table-Based Attacks [64]	x86	CR0.CD	Stealing data from Intel SGX enclave [19].	✓
Super Root Attacks [79]	ARM	DBGBCR, HDCR, HVC	Obtaining the kernel or the hypervisor privilege.	✓
SgxPectre Attacks [16]	x86	MSR 0x48, MSR 0x49	Stealing attestation keys of Intel SGX.	✓
TRESOR-HUNT Attacks [15]	x86	DR0-7	Stealing cryptographic keys.	✓
Voltage-based Attacks [36, 48, 54]	x86	MSR 0x150	Injecting bit flip to / Stealing secret from Intel SGX enclave.	✓

Attacks By Abusing ISA Resources: The ISA resources are often used for system control, including memory mapping, exception handling, virtualization, and many others. Once the ISA resources are abused, attackers can perform various attacks [15, 36, 48, 51, 54, 64, 77, 79]. Table 1 shows some ISA-abuse-based attacks, whose prerequisite is access to certain ISA resources. For example, x86 processors allow system software to configure the CPU frequency and voltage for power management. The $MSR\ 0x150$ is a register for configuring the frequency and voltage. The attackers can use this register to perform various attacks, including injecting fault to enclave applications [48], tampering with the secure memory of SGX [36], and stealing sensitive data from enclave applications [54]. Another example is the cycle counter instructions (e.g., the *rdtsc* in x86) and cache maintaining instructions, which the attackers can use to speed up the timing-based side-channel attacks. Besides the above examples, many ISA resources are critical for the system, and tampering with them could directly break the system-level security properties. For example, the memory mapping is controlled by the page table base address register (e.g., *CR3* in x86 and *SATP* in RISC-V). Once such a register is abused, attackers can construct malicious mappings and break the page table isolation.

Importance of ISA Resources Access Control: Controlling the privilege of instructions and registers is necessary to enhance system security. Hardware manufacturers have acknowledged that manipulating certain hardware features causes security problems. As a result, CPU vendors have implemented mechanisms that control the access privilege of specific ISA resources. For instance, to mitigate the above attacks based on CPU frequency and voltage, Intel adds patches in BIOS and CPU microcode to allow users to configure whether the software can access $MSR\ 0x150$. Additionally, for *rdtsc*, the *CR4* register on x86 specifies whether this instruction can be executed in user space. In ARM, the cache flush instructions can be disabled in user space. Meanwhile, Modern processors introduce privilege levels to manage access privilege for instructions and registers at a coarse granularity.

Memory Isolation Needs ISA Resources Access Control: There exist many systems for memory access control. However, more than memory isolation is needed to build a secure system. For example, the Intel MPK is used by many systems for isolating user-level memory, but it has the security problem of abusing the *wrpkru* instruction. Such an instruction is used to modify the *PKRU* register and change memory access permission. However, *wrpkru* can be executed by any user-space code. Untrusted code can directly execute *wrpkru* and switch to an arbitrary memory domain, which breaks the memory isolation provided by MPK. Thus it requires methods to deprive the untrusted code of the capability to execute *wrpkru*. Similarly, the Intel PKS [35], which provides memory isolation in kernel space, also needs to control the privilege of *wrpkrs* instruction. Besides MPK, many software-based memory isolation systems also require access control of ISA resources. Nested Kernel [21] separates the privilege of the kernel into two parts: a trusted inner kernel controlling the mapping and a de-privileged outer kernel containing other functionalities. It must enforce that the outer kernel cannot modify *CR0*, *CR3*, *CR4* or other memory management registers. Meanwhile, PrivBox [38] and Colony [76] also require untrusted privileged components cannot access privileged ISA resources. For all the above systems, access control of ISA resources is indispensable.

Vulnerability Mitigation Needs ISA Resources Access Control: Also, if the system can control the access privilege of ISA resources with fine granularity, ISA-abused-based attacks can be mitigated more efficiently. With the access control of ISA resources, the developer can selectively expose instructions/registers to a kernel component. Thus the exploit in a module can only make limited privileges available to the attacker. This is actually how the least privilege principle works for mitigation. Take the voltage-based attack [36] as an example. If there is a vulnerability that the attacker can exploit to execute *wrmsr* (write MSRs in x86) in an unrelated kernel module (e.g., debug module), then the attacker can exploit it and write the $MSR\ 0x150$ to mount the voltage-based attack. On the contrary, with ISA resources access control, the kernel can only give the debug module privileges to access the debug registers. Thus, even if the debug module is exploited, $MSR\ 0x150$ can never be accessed by the attacker. How the other attacks are mitigated by the access control of ISA resources is discussed in Section 8.

2.3 Limitations of Current ISA Access Control

Hardware Approaches: Modern processors mainly rely on the CPU privilege level (called exception level in ARM) to perform access control of ISA resources. However, since the CPU only provides limited privilege levels (often including the user level, kernel level, and hypervisor level), the software in a single privilege level can access many instructions and registers. The software in the kernel level can access most ISA resources. Once there is a bug in the kernel level, attackers can further access all ISA resources in kernel space and user space. Unfortunately, one privilege level usually contains a large amount of code, corresponding to many vulnerabilities. Any of these vulnerabilities may give attackers the ability to access critical ISA resources. Because of that, the coarse-grained access control provided by the CPU privilege level is not enough for controlling ISA resources. There also exist other hardware extensions such as CODOMS [65] and Mondrix [73]. But they are still too coarse-grained. CODOMS [65] designs a new hardware extension and uses it to isolate Linux. Mondrix [73] can choose whether to give a domain kernel privileges or not. However, in either of the two works, all the kernel privileges for ISA resources are bound together and are indicated by only one bit for each domain. The result is that a considerable portion of the code still runs in domains with complete kernel privileges.

Virtualization Approaches: Virtualization extensions in modern processors support trap-and-emulate. The privileged instructions cause traps to the hypervisor. The hypervisor can then check if this instruction is allowed in the current privilege domain. However, the traps from virtual machines take many cycles (even an empty VM call takes about 1700 cycles [29]), and the privilege check of the instruction after the trap is not optimized by hardware. Moreover, only hardware-specified privileged instructions can be checked via this approach, which can lead to limitations in cases such as restricting the usage of Intel MPK or PKS instructions, as these instructions do not trap, and are therefore excluded from the scope of virtualization.

Software Approaches: Existing systems based on Intel MPK [27, 29, 62] have to solve the problem of abusing *wrpkru* instruction. ERIM [62] tries to use binary scanning and rewriting to remove unintended *wrpkru* instructions. This is complex because directly rewriting the binary requires code disassembling. And the new instruction sequence may be longer than the original. Thus the control flow instruction must also be rewritten. The correctness and completeness of the rewriting strategy are not proved by ERIM. Even worse, such rewriting is inapplicable in the general case because of undecidable instruction alignment [55, 69]. Hodor [29] also uses binary scanning, but it adds hardware breakpoints to a page's unintended *wrpkru*. If the number of *wrpkru* in a page exceeds the number of debug registers, the code page uses single-step execution. These pages are unmapped, and the fault handler adds breakpoints or starts single-step execution when these pages are accessed. However, even after designing the system carefully, these solutions based on MPK are still error-prone and fail in some corner cases [18, 66]. Thus recent works have to add more patches for enhancement continually [58, 66].

Nested Kernel [21] also uses a software approach to de-privilege the outer kernel. First, sensitive instructions are unmapped from the kernel and mapped back when used. Second, to defend against ROP (return-oriented programming) attacks, Nested Kernel must enforce that no code gadget can be chained together to construct sensitive instructions. So it requires the developers to analyze the compiled binary and manually modify the source code by adjusting alignments, changing functions, and adding *nops* to eliminate implicit sensitive instructions. When the kernel is loaded, it scans the binary and rejects the code containing sensitive instructions. However, manually finding and modifying all possible gadgets is error-prone and significantly increases development effort [56, 66]. And it is extremely difficult to control generic ISA resources [56]. For example, the x86 *out* instruction is only one byte and appears over 50k times in Linux v5.4 kernel image, while only 300+ of them are intended. It is almost impossible to remove unintended occurrences of such instructions manually.

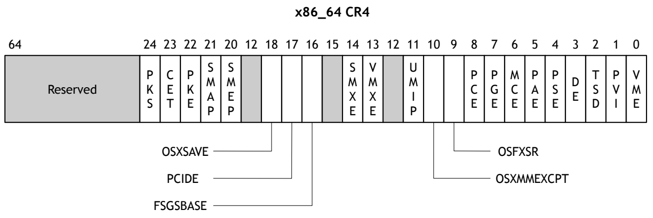


Figure 1: CR4 register in x86_64. Each field represents a hardware capability.

3 SYSTEM OVERVIEW

3.1 System Goals

The main goal of ISA-Grid is to propose a hardware extension to control access privilege for instructions and registers. The detailed goals are:

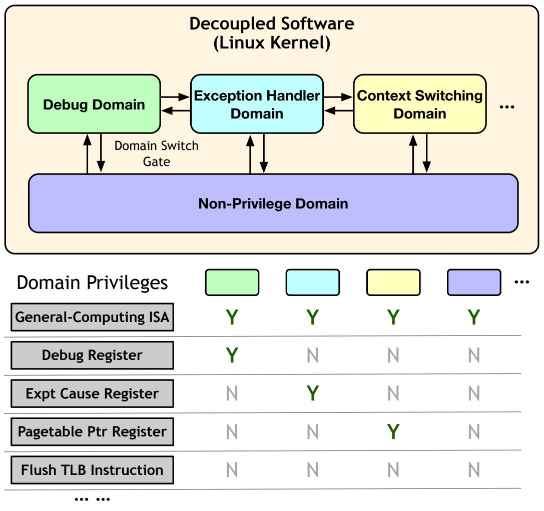
- **Fine-Grained Control**: ISA-Grid should offer access control at the instruction and register level. For registers containing multiple function fields, ISA-Grid should support bit-level access control.
- **Flexibility**: ISA-Grid should allow multiple privilege domains with varying access privileges. The number of privilege domains should be sufficient to isolate different software components.
- **Security**: ISA-Grid must ensure that hardware privilege control cannot be bypassed, so that software running in a privileged domain cannot access ISA resources without permission. ISA-Grid must also prevent a domain from arbitrarily switching to other domains.
- **Performance**: ISA-Grid should only have limited performance overhead on protected software.

3.2 Design Challenges

There are three main challenges for designing a fined-grained hardware access control mechanism for ISA resources.

Challenge-I: Access Control Granularity. The ISA resources have significantly different requirements of control granularity. For instructions and part of registers, each of them can be controlled as a non-divisible entity. However, many system control registers contain multiple function fields. For example, the *CR4* register in x86 contains 20+ different fields, representing different hardware capabilities (as shown in Figure 1). Each bit of the register has a distinct function and may be accessed by different software components. These registers require bit-level access control. ISA-Grid should support such a hybrid granularity.

Challenge-II: Secure Domain Switching. In addition to controlling access privileges for each domain, secure domain switching is crucial. Existing memory isolation mechanisms leverage a higher privilege level to perform domain switching (e.g., the page table isolation schemes rely on the kernel-level code to switch memory mappings) or ask the software developer to carefully design a switch function to enforce the switching security (e.g., Intel MPK relies on the developer to design a secure domain switching gate [29, 62]). Both methods have drawbacks: changing to a



Domain Privileges	🟩	🟦	🟨	🟪	...
General-Computing ISA	Y	Y	Y	Y	
Debug Register	Y	N	N	N	
Expt Cause Register	N	Y	N	N	
Pagetable Ptr Register	N	N	Y	N	
Flush TLB Instruction	N	N	N	N	
... ...					

Figure 2: Decoupled software (Linux kernel) with ISA-Grid abstraction.

higher privilege level increases performance overhead when domain switching is frequent, and the latter method does not guarantee strong security [18]. ISA-Grid should provide a secure and fast method for domain switching.

Challenge-III: Low Performance Overhead. To enforce access control of ISA resources, ISA-Grid needs to check all executed instructions. The overhead of the privilege check might affect the overall performance of applications. ISA-Grid should achieve low overhead for both the privilege check and the domain switching.

3.3 Overview of ISA-Grid

ISA-Grid Abstraction: We first introduce the hardware abstraction provided by ISA-Grid. ISA-Grid performs access control of ISA resources based on **ISA domains**. Each ISA domain can be given privileges to access different instructions and registers. One CPU core only runs in one ISA domain at a time. ISA-Grid enforces that software can only access ISA resources specified by the current ISA domain. As shown in Figure 2, a software program (e.g., the Linux kernel) can be decoupled into different ISA domains with different ISA privileges. ISA-Grid introduces new instructions for switching between different domains.

Design Overview: ISA-Grid introduces a new hardware unit called PCU (Privilege Check Unit) in the CPU core to enforce the privilege control of ISA resources. The PCU is connected to the CPU pipeline to access the required information and then check all executed instructions. If the PCU finds that there is no privilege for the instruction, a hardware exception occurs. As shown in Figure 3, the design of PCU includes three main components: **a Hybrid-grained Privilege Check Engine, an Unforgeable Domain Switching Engine and a Domain Privilege Cache**.

Hybrid-grained Privilege Check Engine: To support privilege control in different granularities, ISA-Grid introduces a hybrid-grained privilege data structure to describe the privileges of ISA resources (**Challenge-I**). For instruction privilege, ISA-Grid leverages the bitmap structure to represent whether an ISA domain can execute

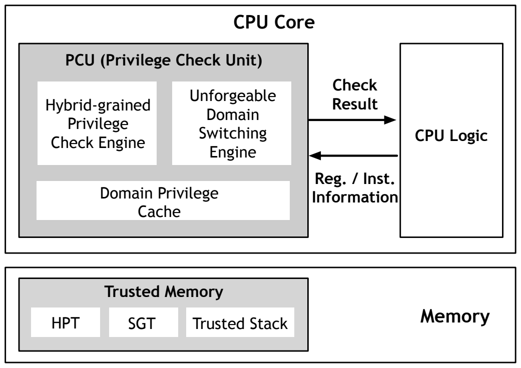


Figure 3: Architecture Overview.

different instructions. Each instruction corresponds to one privilege bit. For register privilege, ISA-Grid uses a double-bitmap as well as on-demand bit-masks to represent whether an ISA domain can access a register and which bits can be accessed. The *hybrid privilege table (HPT)* stores these privilege information for all domains. Such a hybrid privilege data structure allows ISA-Grid to support privilege control of ISA resources with different granularities. More about the hybrid-grained privilege data structure and how to use it for privilege control are shown in Section 4.1. All the privilege data structures are stored in the trusted memory (described in Section 4.5) to protect their integrity.

Unforgeable Domain Switching Engine: ISA-Grid provides an unforgeable domain switching method to enhance security when a domain changes (**Challenge-II**). Such a method provides domain switching instructions for developers and enforces that: 1) a malicious domain switching instructions being injected to the code page cannot be used to switch ISA domains; 2) a malicious switching instruction being constructed by ROP cannot switch domains; and 3) a valid switching instruction cannot be abused to switch to arbitrary domains. ISA-Grid achieves such security goals by allowing software to register multiple unforgeable switching gates. Each of the gates corresponds to a legal switch to an ISA domain. A *switching gate table (SGT)* is used to store all registered gates. With such gates, the domain switching engine can securely switch the ISA domains and change control flow to the target address. Section 4.2 introduces more details.

Domain Privilege Cache: To accelerate the domain switching and privilege check, a domain privilege cache is added in the PCU (**Challenge-III**). It includes an HPT cache and an SGT cache. If such a cache is hit, the privilege check and domain switching can proceed without waiting for memory reading. To reduce the energy cost of the domain privilege cache, ISA-Grid also provides the cache bypass method. More details are described in Section 4.3.

4 SYSTEM DESIGN

This section introduces the detailed design, as shown in Figure 4.

4.1 Hybrid-grained Privilege Check

Each ISA domain has a *unique domain id*. The hybrid-grained privilege check engine inside PCU performs the privilege check. It

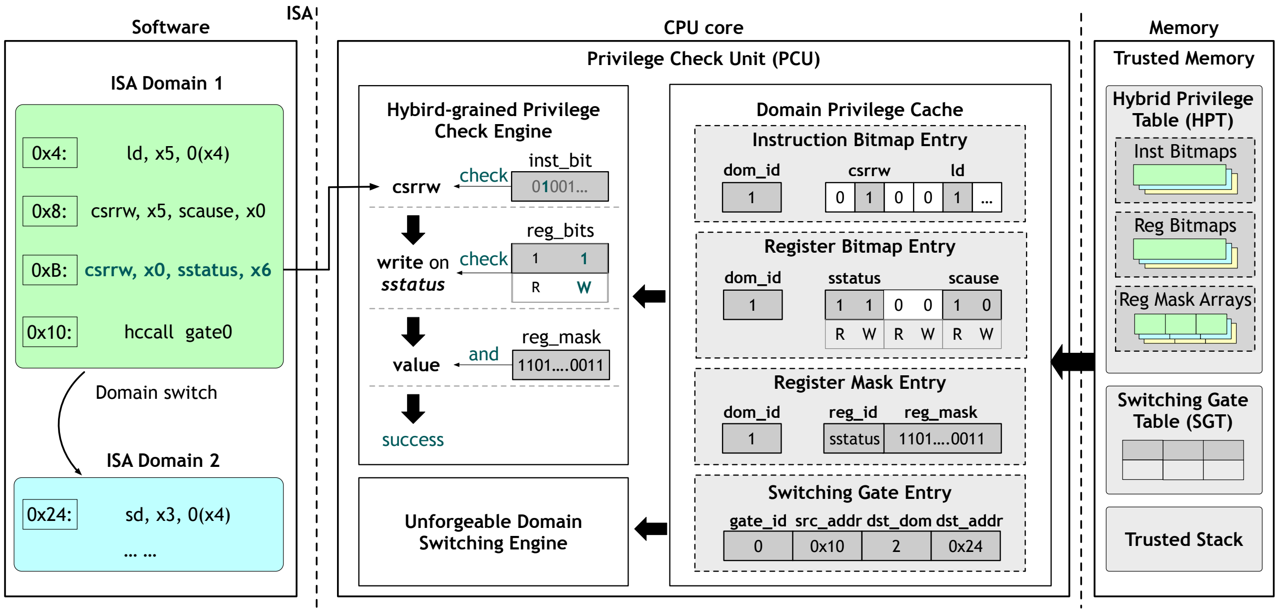


Figure 4: Detailed design of the PCU in ISA-Grid.

leverages the HPT to check all instruction executions and register accesses. The HPT consists of *instruction bitmaps*, *register bitmaps*, and *bit-mask arrays*. For registers, ISA-Grid does not restrict the access privilege of general-purpose registers because they are used for almost all software components. ISA-Grid focuses on the registers for hardware management, which are collectively called Control and Status Registers (CSRs) in this paper. These registers have special use cases and are essential for the entire software stack (e.g., RISC-V *SATP* register). This section describes how ISA-Grid checks access privileges of instructions and CSRs with HPT.

Privilege Check Process: The instructions are checked simultaneously by CPU privilege level and ISA-Grid. An instruction can only be executed when it passes both the check of privilege level and ISA-Grid. Either rejection causes a hardware exception.

When an instruction is issued, ISA-Grid first checks its execution privilege. The instruction bitmap in the HPT is used for the check. If the instruction explicitly accesses the CSRs, ISA-Grid will further perform the register privilege check. ISA-Grid ignores the register privilege check if an instruction accesses the CSRs as its side effect. For example, the RISC-V *ld* instruction can cause exceptions and change the value of the *SCAUSE* register. PCU does not check the register privilege for this instruction. On the contrary, the RISC-V *csrrw* instruction must pass the register privilege check. If the accessed CSR requires bitwise control and is written by the instruction, the bit-mask in HPT is used for the bitwise privilege check. Otherwise, only the register bitmap in HPT is used for the privilege check of the CSR.

Privilege Control for Instructions: In HPT, instruction bitmaps are used for controlling the execution privilege of instructions. Each bit in the bitmap represents whether a particular type of instruction can be executed in an ISA domain. The PCU reads the bitmap to determine the execution privilege of the instructions. The instruction type is specified by the instruction opcode. The hardware maps the opcode of an instruction to the index of the corresponding bit in the bitmap. The length of the bitmap depends on the number of instruction types for a specific architecture.

Privilege Control for Registers: Different from instructions which only require execution privilege, for each CSR, there are two bits in the register bitmap to represent the read and write permission. The address of CSRs, which is used in CSR r/w instructions to specify operands, is mapped to the index in the register bitmap.

Bitwise Control for Registers: Each Control and Status Register (CSR) that requires bitwise access control in ISA-Grid is associated with a bit-mask of the same length for each domain. The bit-mask indicates which bits of the CSR can be modified. Setting a bit in the bit-mask enables modification of the corresponding bit in the CSR. An equation is used to decide if a CSR write operation is permitted. Suppose the original value of the register is V_{CSR}, the value to be written is V_{write}, and the bit-mask is M. Then it should satisfy: $V_{CSR} \oplus V_{write} \wedge \neg M == 0$. The bit-masks are only used for CSR writing and not used for reading. All the bit-masks in a domain constitute a bit-mask array. The CSRs that need bitwise access control are mapped to the indexes in the bit-mask array. The three mappings (instruction type to bitmap index, register address to bitmap index, register address to bit-mask array index) are hardware parameters of ISA-Grid and should be known by software developers.

4.2 Unforgeable Domain Switching

ISA-Grid employs a novel domain switching method that uses dedicated instructions. It has the following properties: **(i):** Each gate can only be called at a fixed address. **(ii):** Each gate jumps to a deterministic address. **(iii):** Each gate changes to a deterministic domain. **(iv):** Unregistered gates can never be executed.

Unforgeable Switching Gate: ISA-Grid uses dedicated gate instructions as domain switching gates. A gate instruction changes the domain and transfers the control flow to the pre-registered target address. The gate instruction itself can be executed by all ISA domains. However, any gate must be registered before it is used. For gate registration, the corresponding entry in the SGT should be configured. Each entry in the SGT contains the gate address, the destination address, and the destination domain of a gate. The entry index of a gate is also used as its gate id. Each gate instruction has to specify its gate id at runtime, which can be stored in a general-purpose register. When a gate instruction is executed, PCU uses the gate id to retrieve the gate information from the SGT. And the hardware checks if the address of the gate instruction is the same as the registered value. If the address matches, the gate instruction changes the control flow to the registered destination address and switches the ISA domain.

Extended Gate: As described above, each gate instruction jumps to a deterministic address and changes to a deterministic domain. If a developer wants to implement a cross-domain function call, there should be a gate instruction for entering the domain and another gate for returning from the domain. Thus traditional call-and-return convention cannot be directly used. To ease the development, ISA-Grid provides extended gate instructions, including an extended gate instruction and an extended return instruction. The registration of the extended gate instruction is the same as the original gate. But when executed, it pushes the return address and source domain id on the trusted stack located in trusted memory. The extended return instruction does not need registration. This instruction pops the return address and domain id from the trusted stack and returns from the cross-domain call. The extended gate and return instructions can be used in pairs for cross-domain calls in a call-and-return manner. With the extended instructions, only one gate needs to be registered for a cross-domain call. For ISA-Grid's implementation, the two extended instructions and the trusted stack are optional.

Analysis: ISA-Grid satisfies **Property (i)** because the address of a gate instruction (unforgeable gate instruction or extended gate instruction) is registered and frozen, and ISA-Grid enforces the equality of the runtime address and registered address whenever a gate instruction is executed. Similarly, because the destination address and ISA domain are registered, the gate instruction cannot be exploited to switch to arbitrary domains or addresses. Thus **Property (ii) and (iii)** are satisfied. If a fake gate instruction is injected at a new address or appears at an instruction boundary, the runtime address of the gate cannot match any entry in the SGT, and this causes an exception. Thus **Property (iv)** is enforced.

4.3 Domain Privilege Cache

ISA-Grid adds a domain privilege cache in PCU to reduce the runtime overhead. The domain privilege cache consists of an HPT cache and an SGT cache. The cache prefetch mechanism can reduce overall cache latency, and the cache bypass mechanism reduces the dynamic energy cost.

HPT Cache: The HPT cache is designed to accelerate access to the hybrid privilege table (HPT), thereby reducing runtime overhead. When there is a cache hit, the checked instruction incurs no extra cycles. On the other hand, when there is a cache miss, the instruction must wait for the corresponding HPT entry to be retrieved from memory. Depending on the implementation, we can stall the instruction by flushing the pipeline or stalling the ROB.

Because there are different privilege structures in HPT, the HPT cache contains three different types of entries. For the bit-mask array, each entry contains a CSR bit-mask. The entry tag consists of the domain id and the CSR index in the bit-mask array. An entry can be retrieved if the tag matches, which means a cache hit. For the register bitmap, each entry contains bits representing the write and read permission for a group of CSRs with adjacent index values. If a single entry contains a bitmap for all the CSRs, it might be quite large and take a long time to fill. So, a register bitmap for a domain can be divided into several entries. For the instruction bitmap, each entry is for a group of bits in an instruction bitmap. The tag of each entry consists of only the domain id and the group id. If requested HPT data is not in the cache, the data are fetched from memory, and the corresponding entry in the cache is filled. All three entry types use the domain id as part of the entry tag. As a result, a cache flush is not necessary when the domain changes. The HPT cache can be implemented either as three separate caches or as a unified cache. In the case of a unified cache implementation, each entry needs an additional field to specify the entry type. A unified cache structure may improve the overall hit rate but incur increased hardware complexity.

SGT Cache: To further reduce the latency of domain switching, an SGT cache is added to PCU to cache the information of recently used gates. The tag of each entry is the gate id, and the payload of each cache entry is an entry in SGT. Given the gate id, the SGT entry for the requested gate could be directly retrieved from the cache. If the requested gate is not in the cache, the corresponding SGT entry is loaded from memory and inserted into the cache.

Cache Bypass For Saving Energy: PCU checks every instruction to be executed. However, the PCU needs to look up the cache to get the entry containing the instruction bitmap before the instruction privilege check. Thus, unlike CSRs, the instructions' execution privilege is checked more frequently. ISA-Grid uses a cache bypass mechanism to decrease the dynamic energy cost. ISA-Grid introduces an instruction privilege register to store the instruction bitmap of the current domain. The register is filled with the current domain's bitmap when the current domain's first instruction is executed. And then, the PCU can directly use the bitmap in this register for the instruction privilege check. So the instruction bitmap cache is only looked up fixed times after a domain switching. This mechanism reduces fully associative cache lookups and decreases the dynamic energy cost.

Cache Prefetch: A CSR r/w instruction might cause a long latency because of memory access if the corresponding entries are not in the HPT cache. ISA-Grid provides software prefetch mechanisms to load CSR bitmaps or masks into the HPT cache in advance. Sometimes only a few CSRs are accessed before exiting the current domain, and prefetching all the CSRs is unnecessary. The software developer can choose to fetch entries for all the CSRs or just certain CSRs using the dedicated prefetch instruction. The memory access requests from prefetching have a lower priority than the normal memory requests. Thus, the cache prefetch does not significantly affect the latency of normal load and store instructions. Section 5.1 introduces more about the prefetch instruction.

4.4 Special Domain for Initialization

As mentioned before, each ISA domain has a unique id. Upon reboot, the domain id for the processor is reset to zero, which corresponds to the special initialization domain, domain-0. This domain is given all the privileges by default and is responsible for initializing ISA-Grid. After initialization is complete in this domain, the supervisor can switch to a new domain and do real jobs. Domain-0 can also be used for configuring new domains and gates after initialization. The code in domain-0 should be kept small. The domain-0 can only be entered when the processor reboots or executes the gate instructions. The extended return instruction described in Section 4.2 is not allowed to return to domain-0. Thus the instruction can not be exploited to switch to domain-0 with all the privileges and a non-registered destination address.

4.5 Trusted Memory

ISA-Grid needs a region of trusted memory to store the SGT, HPT and the trusted stack. There are different ways to design such a memory region. ISA-Grid uses a clean and architecture-independent design for trusted memory. A special range of physical memory is reserved for trusted memory. Two dedicated registers specify the range and are set in domain-0, and every memory load or store instruction is checked. The load and store instructions can access the trusted memory region only in domain-0. In other ISA domains, this memory region can only be accessed by PCU, and using load or store instructions to access the trusted memory causes exceptions. The bound check overhead can be minimized by forcing the trusted memory to be power-of-two sized and aligned. This design of trusted memory can be integrated into different ISAs. For the trusted memory, there also exist other design choices. For example, it can be integrated into existing memory protection mechanisms such as RISC-V Physical Memory Protection (PMP) [70].

5 SOFTWARE INTERFACE

5.1 ISA Extension

We introduce the ISA extensions of ISA-Grid in this section. New registers and instructions are summarized in Table 2.

Domain Setup: There are four different registers that store the address of the in-memory structures of ISA-Grid. The *csr-cap*, *csr-mask*, *inst-cap*, and *gate-addr* store the base address of register bitmaps, bit-mask arrays, instruction bitmaps, and SGT. A *domain* register is used to store the current domain id. ISA-Grid can support at most 2^{64} domains in the current design. Only domain switching instructions can change the *domain* register. The read permission of the *domain* register can be configured by ISA-Grid, but normal CSR write instructions cannot change it.

Domain Switching: ISA-Grid provides *hccall* as the basic gate instruction and *hccalls* as the extended gate instruction. The *hcrets* is the extended return instruction. The trusted stack is in the trusted memory and is bounded by *hcsb* and *hcsl* register, which are the base and limit of the stack. The stack pointer is *hcsp* and can be changed by *hccalls* and *hcrets*. The *hcsp* values outside the range of *hcsb* and *hcsl* cause exceptions.

Cache management: The *pfch* instruction is used for cache prefetching. The prefetch instruction makes the PCU load bitmaps and masks from memory to fill cache entries of requested CSRs in the current domain. The *pfch* can fetch entries of all the CSRs or certain CSRs depending on the operands. The *pflh* instruction is used for flushing the cache. Different modules of domain privilege cache (described in Section 4.3) are given different ids, and the *pflh* can use the cache id to choose which cache to flush.

5.2 Programming Model

Domain & Gate Registration: The processor reboots with domain-0, and then ISA-Grid needs to be initialized first. The *csr-cap*, *csr-mask*, and *inst-cap* should correctly point to structures in HPT. An SGT should be created in trusted memory, and the base address should be stored in *gate-addr*. The process of creating a new domain or gate involves adding entries in these in-memory structures. HPT and SGT are stored in the trusted memory, so only domain-0 can configure new domains and gates. The registration of gates and domains can be performed either during the system boot or at runtime.

During system boot, domain-0's software creates a special gate for registering new domains and gates. Such a gate switches to domain-0 and jumps to the entry of a domain-0 function which creates domains and gates. The gate is valid for all kernel components, allowing them to invoke it for registration. Domain-0's software also creates a basic domain for kernel execution and a gate to enter the basic domain. The gate is used for leaving domain-0 for the first time when other domains are not yet created. After initialization is finished and the CPU leaves domain-0, the kernel components can invoke domain-0 to register domains and gates. How to pass the required arguments for a domain/gate registration, including privileges of a domain or the source/destination of a gate, depends on the software design. Domain-0's software can use these arguments to configure the HPT and SGT and return the domain/gate id.

Although domains can be added at runtime, there is no conflict between domains. ISA-Grid does not force the privileges of different domains to be mutually exclusive. However, developers could implement a policy in domain-0 to reject creating domains with overlapping privileges. Also, ISA-Grid works even if KASLR is enabled. It is because the domains and gates are registered after the kernel or kernel modules are loaded. After loading, the address of the kernel code is already determined.

Cross-domain Call: A kernel component can register all its entry functions' information in domain-0. Then, a caller domain can invoke domain-0's software to register a new gate for invoking another domain's entry function. In more detail, the caller passes the address of its gate instruction and the name of the target function to domain-0. Then domain-0 sets up the new gate and allocates the gate id which will be returned to the caller domain. The gate id is further used by the caller domain to call the target domain. The software in domain-0 should check whether a registration request is valid or not. The developer can deploy different security policies in domain-0 to perform such checks.

If only *hccall* is used for domain switching, the trusted stack is not needed. A trusted stack should only be allocated if *hccalls* and *hcrets* are used. The *hcsp*, *hcsb*, and *hcsl* should also be initialized for the trusted stack. The *hccalls* and *hcrets* should always be used in pairs in a call-and-ret manner. After domain changes, *pdomain*

Table 2: New Instructions and Registers introduced by ISA-Grid.

Register	Description (R/W in domain-0.)	Instruction	Description
domain/pdomain	ID of current/previous domain. Read only.	*hccall #gateid*	Domain switching instruction. Check the instruction address. If valid, jump to the
domain-nr	Number of valid domains.		destination address and change domain using *#gateid* stored in a register.
csr-cap	Address of CSR bitmaps.	*hccalls #gateid*	Extended domain switching instruction. Do the same as *hccall*. But push the
csr-bit-mask	Address of CSR bit-mask arrays.		return address and current domain on the trusted stack.
inst-cap	Address of instruction bitmaps.	*hcrets*	Extended domain return instruction. Pop the return domain and return address
gate-addr	Address of SGT.		from the trusted stack. Jump to the return address and change domain.
gate-nr	Number of valid gates.		
hcsp/hcsb/hcsl	Stack pointer/base/limit of trusted stack	*pfch #csr*	Prefetch privilege structures of *#csr*. *#csr* is stored in a register. If *#csr* is zero, fetch all.
tmemb/tmeml	Base/limit address of trusted memory.	*pflh #bufid*	Flush the cache with *#bufid*. *#bufid* is stored in a register. If *#bufid* is zero, flush all.

contains the previous domain id. The register can be read by the code in the current domain for developer-defined security check. The software can maintain separate trusted stacks for different threads. For context switching, the *hcsp*, *hcsb* and *hcsl* of a thread must be saved in trusted memory and restored later. This save-and-restore procedure can be done in domain-0.

6 USE CASES

ISA-Grid enhances security by providing instruction-level and bit-level access controls for instructions and registers, thereby improving the principle of least privilege. With ISA-Grid, developers can selectively grant the necessary privileges for a given software component based on the required ISA operations. This level of permission granularity allows a software component to be configured with privileges to execute a single privileged instruction or to access a single bit of a privileged register.

6.1 Linux Kernel Decomposition

We decompose the Linux kernel in a new way with ISA-Grid on both the RISC-V and x86 prototypes. The new decomposition scheme is based on the ISA resources used by the code and can be further combined with memory isolation to provide stronger isolation. The isolation scheme can perform as an efficient mitigation for ISA-abuse-based attacks.

We introduce the decomposed kernel with the x86 prototype here. In the kernel, some registers are only used for system initialization. And values of these registers are already frozen before the initial process runs, such as *IDTR*, *GDTR*, most bits in *CR0*/*CR4*, and most of the MSRs. Thus, we deprive the kernel of the ability to write *IDTR*, *GDTR*, and most of the bits in *CR0*/*CR4* after the kernel finishes the initialization. We analyze the MSRs supported by the Gem5 simulator in kernel code and put each of such functions that modify any of the MSRs in a different ISA domain with only the privilege of modifying the MSRs involved in the function. Also, each function that modifies any of *LDTR*, *CR0.TS*, *CR0.NE*, *CR3* or *CR4.SMAP* is put in a different ISA domain that only allows the modification of registers or bits involved in the function. The other code runs in a domain that cannot modify control registers and MSRs. In our implementation, the domains are manually decided, but it is feasible to add specialized plugins to the compiler to facilitate domain division. We assume that the attacker is a user seeking to mount some ISA-abuse-based attacks (e.g., attacks in Table 1). Thus the attacker aims to exploit kernel vulnerabilities to access certain registers or execute certain instructions in order to launch attacks. With ISA-Grid, the privilege of writing CSRs is only given to specific function. Therefore, in other ISA domains, the kernel cannot execute the disabled instructions even if such instructions appear at the instruction boundaries, mixed read-only data in the text segment, or injected code. Thus a vulnerability in one component with no privilege of modifying a register never gives the attacker chances to mount ISA-abuse-based attacks related to that register. And the ISA-abuse-based attacks can be mitigated.

6.2 Enhancing Nested Kernel

To show how ISA-Grid can be integrated into memory protection mechanisms to enhance system security, we combine ISA-Grid with Nested Kernel [21]. With ISA-Grid, the memory protection mechanism is more secure and practical to use.

The malicious access of the *CR0*, *CR3*, *CR4*, *IDTR*, or *MSR EFER* can break the memory protection properties of Nested Kernel. As mentioned in Section 2.3, Nested Kernel relies on the developer to manually change the kernel code to remove the unintended sensitive instructions from compiled binaries, which is error-prone and impractical. Nested Kernel rejects kernel modules that contain unintended sensitive instructions, including those appearing at the instruction boundaries, even if the module is correctly implemented. With ISA-Grid, the hardware enforces that the unintended sensitive instructions are never executed, and therefore the module can be loaded. We use ISA-Grid to construct a nested monitor with the same ability as Nested Kernel. Various security policies such as write-once data can be applied [21]. The nested monitor runs in an ISA domain with the privilege of writing the MSRs and control registers, while the outer kernel is in a domain that cannot modify these registers except for *CR4.SMAP*. Except for the switching of ISA domains, the entry and exit gates of the monitor are the same as Nested Kernel. The entry gate uses *hccall* to switch to the monitor's ISA domain. It then sets the *CR0.WP*, does other checks and jumps to the monitor code. The exit gate clears the *CR0.WP* and switches to the outer kernel's ISA domain and code.

6.3 Emerging Hardware Feature

ISA-Grid can enhance memory isolation provided by the new hardware feature, Intel Protection Key for Supervisor (PKS) [35]. Just like MPK-based works [29, 62], the malicious execution of the *wrpkrs* instruction can break the memory isolation provided by PKS. However, as mentioned in Section 2.3, existing software approaches are vulnerable. ISA-Grid provides a more reliable hardware approach to disable *wrpkru* in untrusted code and introduces very limited overhead. With ISA-Grid, the trampoline function that modifies the *PKRS* register can run in an ISA domain where *wrpkrs* can be executed, and the other code runs in an ISA domain where *wrpkrs* is disabled. Then the *wrpkrs* can never be executed outside the trampoline function.

6.4 Other Use Cases

The fine-grained ISA access control of ISA-Grid can also be used in the sandbox or virtualization systems. PrivBox [38] proposes an in-kernel sandbox to run application code. The sandbox needs to prevent the inside code from executing privileged instructions. Colony [76] constructs software TEEs (trusted execution environments) with a trusted monitor and has to ensure that there are no critical instructions outside the trusted monitor. With ISA-Grid, the execution of privileged instruction can be eliminated more reliably in these two works. Dune [12] allows Dune Processes leveraging libDune to run in VMX non-root ring 0 to access privileged resources for security and performance. The Dune process can access most of the privileged ISA resources, which is dangerous when the code is vulnerable. With ISA-Grid, the Dune Processes and libDune can be divided into ISA domains, reducing the chance of exploitation.

7 EVALUATION

The evaluation tries to answer the following questions: 1) How effective is the domain switching of ISA-Grid? 2) What is the runtime overhead of ISA-Grid? 3) What is the hardware resource cost of ISA-Grid?

Configuration: In the prototype, the HPT cache is implemented as three separate caches. Thus, ISA-Grid introduces three HPT caches and one SGT cache, and each of them is implemented as a fully associative LRU cache. The evaluation tests ISA-Grid with three configurations. The **16E.** and **8E.** mean each cache has 16 or 8 entries. The **8E.N** uses eight entries for each HPT cache but no SGT cache.

RISC-V Prototype: We implement the RISC-V prototype based on Rocket Core [9], which is a 5-stage in-order scalar processor. The Privilege Check Unit (PCU) is implemented as a unit of the Rocket Core. The *SSTATUS* register needs bitwise control. Other supervisor and user CSRs only require the check with register bitmaps. The prototype implementation only supports 2^{12} domains to reduce the cache entry size. The modified RISC-V core runs on the Xilinx VC707 FPGA board at 100MHZ with 1GB DDR3 memory.

x86 Prototype: The x86 prototype is implemented based on the Gem5 [13] cycle-accurate simulator, and the O3 core model is used. In this prototype, the *CR0* and *CR4* need bitwise control. Other control registers and MSRs only need read and write control. Some x86 instructions can be combined with different prefixes. The ISA-Grid ignores the instruction prefix and use the opcode to decide the instruction type. Table 3 shows the detailed configuration of the simulated x86 processor.

Software Setup: We use **Linux 5.15** for the RISC-V prototype and **Linux 5.4** for the x86 prototype. **LMbench** [46] is used to measure low-level OS operations. Also, some real-world applications are used. **SQLite3** [3] is a widely-used database engine. We use the speed benchmark tool of SQLite3. **Mbedtls** [8] is a C library that implements cryptographic primitives and the SSL/TLS protocols.

Table 3: Simulation Parameters.

HW	Parameter
Pipeline	8 fetch/decode/issue/commit, 32/32 SQ/LQ entries, 192 ROB entries, Tournament branch predictor
L1 I-Cache	32 KB, 64 B line, 4-way, 2-cycle latency, 4 MSHRs
L1 D-Cache	32 KB, 64 B line, 4-way, 2-cycle latency, 4 MSHRs
L2 Cache	256 KB, 64 B line, 16-way, 20-cycle latency, 20 MSHRs
L3 Cache	2 MB, 64 B line, 16-way, 32-cycle latency, 512 MSHRs
Memory Latency	30ns after cache miss

Table 4: Domain switching latency. (* means ISA-Grid)

CPU	Instruction	Cycles	Explanation
RISC-V Rocket [9]	load/store	>120	Cache miss latency.
*** RISC-V Rocket**	**hccall**	**5**	**Gate instruction.**
*** RISC-V Rocket**	**hccalls/hcrets**	**12 / 12**	**Extended gate/return inst.**
x86 Gem5	load/store	>200	Cache miss latency.
*** x86 Gem5**	**hccall**	**34**	**Gate instruction.**
*** x86 Gem5**	**hccalls/hcrets**	**52 / 44**	**Extended gate/return inst.**
CPU	**Scheme**	**Cycles**	**Explanation**
CHERI MIPS	CHERI [71]	>400	Change capability for memory.
RISC-V Ariane [1]	Donky [59]	2136	Change memory permission.
RISC-V Rocket	System call	532	Empty call w/ PTI.
RISC-V Rocket	Supervisor call	434	Empty supervisor call.
*** RISC-V Rocket**	**X-domain call**	**13 / 32**	**Empty call (*hccall* / *hccalls*).**
Intel i7-4770	x86 Rings [39]	>1300	Call to unused privilege levels.
x86 Gem5	System call	1050	Empty call w/ PTI.
*** x86 Gem5**	**X-domain call**	**70 / 87**	**Empty call (*hccall* / *hccalls*).**

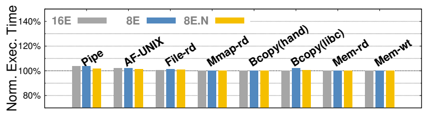
We use the benchmark tool provided by Mbedtls. For **file compression**, we use the *gzip* command to measure the extraction and compression of the kernel image of Linux 5.15. Also, we use the *tar* command to measure the extraction and compression of the Mbedtls-3.1.0 source code.

7.1 Microbenchmarks

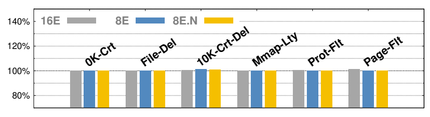
We evaluate the latency of ISA-Grid's domain switching instructions (Table 4). We compare ISA-Grid's domain switching with the empty system calls, empty supervisor calls (RISC-V only), and other isolation mechanisms' domain switching operations. The **X-domain call** means an empty function call with ISA domain switching. We evaluate the latency with two *hccall* and with *hccalls* + *hcrets*. The empty **X-domain call** on x86 is faster than the sum of *hccalls* and *hcrets* because of the store-to-load forwarding in the load-store queue. We evaluate the cache hit rate on the **8E.** x86 prototype with the decomposed Linux kernel described in Section 6.1. We run the three applications with the decomposed kernel and record the cache hit rate. After the applications run, the cache hit rates of all the SGT caches and HPT cache reach 99.9%. It is because some functions in the kernel are very hot when user applications are running. ISA-Grid can be used for different purposes, and the hit rate might differ depending on the workload.

7.2 Use Case Evaluation & Analysis

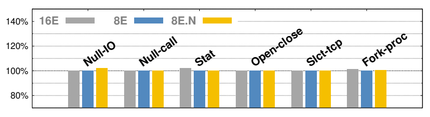
Case 1: Linux Decomposition. We implement the Linux decomposition use case (Section 6.1) with both the RISC-V and x86 kernels. We run LMbench [46] on the RISC-V kernel and three real-world



(a) LMbench - Local Communication latencies

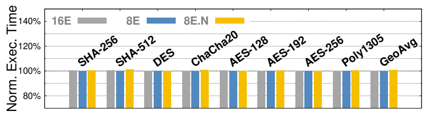


(b) LMbench - File and VM system latencies

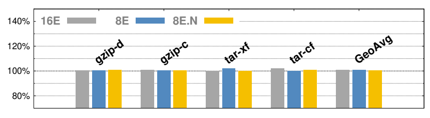


(c) LMbench - Processor, Processes

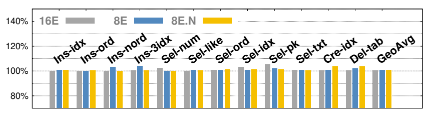
Figure 5: Normalized execution time for LMbench benchmarks with case 1: Linux decomposition on RISC-V.



(a) Mbedtls Encryption



(b) File Compression



(c) SQLite Benchmark

Figure 6: Normalized execution time for different applications with case 1: Linux decomposition on RISC-V.

applications on both kernels. The results of LMbench and applications are shown in Figure 5, 6 and 7. We run these benchmarks multiple times and calculate the average overhead to reduce the error of randomness. The results show that decomposing Linux with ISA-Grid only causes limited overhead (**less than 1%** for real-world applications). The SQLite benchmark with FPGA has more noise with even an unmodified kernel and hardware because of the latency and throughput variation of the external storage card.

Case 2: Enhancing Nested Kernel. We implement the Nested Kernel use case (Section 6.2) on x86 with **8E.**. We use three applications to evaluate the performance of the kernel with a nested monitor. The performance is compared with the unmodified Linux kernel. Compared to the original implementation of Nested Kernel [21], our implementation does not need to scan the binary or unmap the code that modifies the control registers and MSRs. We evaluate two different implementations. **Nest.Mon.** uses a monitor that mediates all the memory mapping changes, and **Nest.Mon.Log** uses a monitor that mediates all the changes and logs recent modifications of page tables with a circular buffer. The results are shown in Figure 8. The overhead of different applications is lower than 1%.

Case 3: Emerging Hardware Feature. The performance of Intel PKS [35] can be estimated with Intel MPK. We add the ISA domain switching overhead to a trampoline function that changes memory permission with *wrpkru* and calls the target function. We estimate the overhead of domain switching with both PKS and ISA-Grid using metrics from Hodor [29]. The *wrpkru* takes 26 cycles, and a trampoline function with MPK domain switching takes 105 cycles. Switching to an ISA domain where *wrpkrs* is enabled and back with two *hccall* needs 70 cycles. If the *wrpkru* and *wrpkrs* instructions have the same latency, domain switching with both PKS and ISA-Grid takes 105 + 70 = 175 cycles. It is still faster than other methods of changing memory permission (938/577 cycles for changing page table with/without page table isolation, 268 cycles for changing extended page table using *vmfunc* [29]).

Case 4: Multiple Service Protection. We use four simple services, implemented as different kernel modules, to evaluate the performance overhead of protecting multiple services on x86 Linux. Each of them is put in a different ISA domain to access different privileged ISA resources at the same time. These services use CPUID instruction, memory type range registers (MTRRs), or performance-monitoring counters (PMCs). A user application invokes these services with *ioctl* and evaluates the latency. The baseline is directly running services with unmodified Linux. As shown in Table 5, ISA-Grid only causes less than 5% overhead. A real-world service often contains more complex logic, and the overhead of ISA-Grid should be much smaller.

Table 5: Latency for different services (in cycles).

	Inst./Reg.	Purpose	ISA-Grid	Native	Overhead
Service-1	CPUID	Get CPU information.	2081	1997	4.21%
Service-2	MTRR	Get memory type.	2038	1970	3.45%
Service-3	PMC	Get number of interrupts.	1803	1721	4.76%
Service-4	PMC	Get number of iTLB miss.	1776	1698	4.60%

Table 6: Hardware cost of ISA-Grid

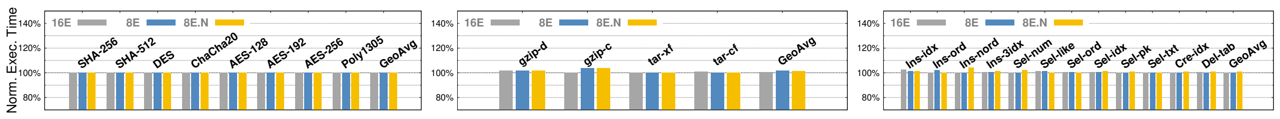
	Rocket Core	16E.	8E.	8E.N
LUT as Logic	51137	53421(4.47%)	52685(3.03%)	52267(2.21%)
LUT as Memory	6420	6420(0%)	6420(0%)	6420(0%)
Slice Registers	37576	40280(7.20%)	39208(4.34%)	38683(2.95%)
RAMB36	10	10(0%)	10(0%)	10(0%)
RAMB18	10	10(0%)	10(0%)	10(0%)
DSP48E1	15	15(0%)	15(0%)	15(0%)

7.3 Hardware Resource Cost

We use Vivado [4] for synthesis and hardware implementation for FPGA and use the report of Vivado to analyze resource utilization. ISA-Grid uses more LUT and Slice Registers than the original Rocket Core, as shown in Table 6. The hardware cost is mainly from different kinds of caches. If ISA-Grid uses **8E.N**, the hardware cost is limited (2.21% of LUT, 2.95% of Slice Register, and 0.0% of RAM and DSP). The utilization can be further optimized by adjusting cache size and using Verilog instead of Chisel.

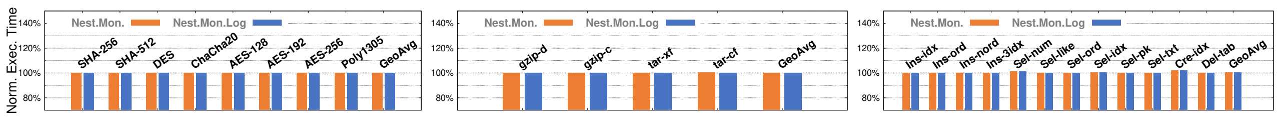
8 DISCUSSION

Security Analysis: ISA-Grid can control the privileges of ISA resources in a fine-grained manner and give only a necessary subset of privileges to a software component. Leveraging such fine-grained



(a) Mbedtls Encryption (b) File Compression (c) SQLite Benchmark

Figure 7: Normalized execution time for different applications with case 1: Linux decomposition on x86.



(a) Mbedtls Encryption (b) File Compression (c) SQLite Benchmark

Figure 8: Normalized execution time for different applications with case 2: Nested Kernel on x86.

access control, even if a vulnerability is exploited, only the privileges assigned to the compromised component can be exposed to the attacker. All the ISA-abuse-based attacks listed in Table 1 can be mitigated by ISA-Grid like cases in Section 6. For the Controlled-Channel Attacks [77], the access privilege of *IDTR* can be limited to the component that initializes the interrupt descriptor table (IDT) by ISA-Grid. Thus the exploitation in another component cannot modify *IDTR*. And the potential attack is prevented. For the Stealthy Page Table-Based Attacks [64], the *CR0.CD* is not allowed to be changed during normal kernel execution. Thus the vulnerabilities cannot cause the modification of *CR0.CD*. And this attack is mitigated. For the TRESOR-HUNT/FORESHADOW Attacks [15, 63] or NAILGUN Attacks [51], the x86 debug registers or ARM PMU registers can only be made available for the trusted kernel components and only exposed interfaces to trusted services. For the Super Root Attacks [79], the access of related registered can be disabled in unrelated components. For the Voltage-based Attacks [36, 48, 54], the *MSR 0x150* is not allowed to be changed outside the initialization code of the kernel. There is no chance for the attackers to change *MSR 0x150* after initialization. Thus the exploitation in the kernel or the untrusted applications cannot mount such attacks. For SgxPectre Attacks [16], some BTB features can be set in *MSR 0x48* and *MSR 0x49* during initialization, and then the modification of these registers can be disabled by ISA-Grid. Then such attacks are mitigated. And for other ISA-abuse-based attacks mentioned in related works [18, 21, 29, 62], they can also be mitigated by limiting the use of some registers and instructions to specified software components such as the cases in Section 6.2 and 6.3. Thus, unintended instructions, such as instructions appearing at the instruction boundaries or constructed at runtime, can never be executed. **For the ISA-abuse-based attacks surveyed in this paper, ISA-Grid can mitigate 100% of these attacks.**

On the other side, the runtime access control provided by ISA-Grid can achieve stronger security than statically checking whether a code component contains illegal instructions. The static method can be bypassed by dynamic code injecting/constructing attacks, e.g., ROP attacks. It is possible to statically discover code gadgets of illegal instructions, but binary rewriting to remove the illegal instruction is inapplicable in the general case because of undecidable instruction alignment [55, 69]. And existing works do not prove the correctness and completeness of such static scanning and rewriting methods targeting specified instructions. ISA-Grid can defend against such dynamic code injecting/constructing attacks.

Development Complexity: First, the configuration of ISA domains is simple. An ISA domain has, by default, no privilege to access privileged instructions/registers, and the code component in the domain needs to acquire the necessary privileges explicitly. And most components do not need additional privileges. Second, the bitmaps/masks-based privilege configuration is not error-prone, which has been used by existing hardware features (e.g., Intel VT). Most developers just need to concern about interaction with domain-0 to ask for privileges and information about cross-domain calls. Finally, developers may need to register domains and add cross-domain calls in their code. This kind of modification is common for existing isolation mechanisms [29, 59, 65, 72], and the complexity is acceptable. Software engineering methods or compiler technologies can be used to simplify development, such as providing SDKs or automation tools. These are works from the perspective of software, which we leave for future work.

Cache Optimization: On the security side, the cache mechanism may be used to mount side-channel attacks (e.g., PRIME + PROBE) to infer what ISA resources are being used. Such information may be sensitive in some scenarios. A domain can make a performance-security tradeoff by flushing the cache before the domain switching to mitigate such attacks. Meanwhile, ISA-Grid uses fully associative caches, which makes it harder to perform cache-based side-channel attacks. More importantly, knowing which instructions or registers are used does not break the security guarantee of ISA-Grid. The hardware still enforces that a domain cannot access the ISA resources without corresponding permissions.

On the performance side, many existing works can be used to optimize the cache performance of ISA-Grid. For example, the method in Draco [60] can be used to reduce the privilege check latency. ISA-Grid can add a cache to store all legal instructions, including the instruction bytecode and register values. If such a cache is hit, the execution is legal, and ISA-Grid does not need to run the privilege check logic.

Possible Simplification: ISA-Grid gives the system the strong capability to control ISA resources in fine granularity. In cases where some instructions/registers are always used together, ISA-Grid with coarser granularity may have worked. For example, RISC-V architecture has many extensions, and customized extensions might provide new instructions/registers that are always used together in a small piece of code. It is possible to simplify the implementation of ISA-Grid by using one bit to control the privilege for a small group of instructions/registers introduced by an extension.

Extending to User Space: Although ISA-Grid focuses on protecting kernel-level software, it can be extended to isolate user-level software. The software in domain-0 needs to: 1) maintain a trusted stack for each user thread and kernel thread, and switch the stack for user-kernel switching and thread switching; and 2) maintain multiple SGTs for different processes and the kernel, and switch among them. While abuse of kernel-level ISA resources can have serious consequences, user-level ISA resources are generally less critical. For this reason, ISA-Grid currently prioritizes the protection of kernel-level software.

9 RELATED WORK

There has been a long line of research on building isolation mechanisms based on various hardware or software platforms. Most of them focus on memory isolation, while ISA-Grid targets ISA resources.

Isolation Using Page Table: Existing works leverage page tables to isolate software components [14, 30, 33, 43, 67, 76]. LWC [43] designs a new OS abstraction for isolation, and the memory isolation is enforced by using different page tables. Wedge [14] could limit memory access and system calls for a thread. SMV [30] could construct secure memory views for different threads. Arbiter [67] designs new memory privilege control interfaces based on page tables for multi-thread applications. Colony [76] constructs isolated execution environments in privileged software by controlling page tables. TZ-Container [33] builds a secure monitor in TrustZone secure world and leverages the page table to isolate container processes.

Isolation Using Virtualization: Virtualization extensions are also used for isolation [32, 41, 44, 47, 49, 53, 68, 78]. CloudVisor [78] uses nested virtualization to isolate different VMs. vTZ [32] leverages ARM TrustZone and virtualization to isolate multiple secure VMs. LXDs [49] uses virtualization to construct isolated domains in kernel space. SeCage [44] uses Intel VT-x to automatically decompose an application and protect the secret data. xMP [53] leverages virtualization extensions to isolate sensitive data for virtual machines. Skybridge [47] uses VMFUNC to speed up the domain switching between isolated environments.

Isolation Using Intel MPK: Recently, there have been many works leveraging Intel MPK to isolate memory resources [25–29, 37, 40, 52, 62]. Libmpk [52] provides virtualization for Intel MPK in order to support more keys. Enclosure [25] and Hodor [29] can use Intel MPK as a backend isolation mechanism. Enclosure restricts untrusted libraries by language construct. Hodor is designed for data plane libraries requiring fast transition. ERIM [62] uses Intel MPK to isolate the trusted and untrusted parts of an application. UnderBridge [27] uses MPK to isolate system servers of microkernels and speed up the IPC. FlexOS [40] allows the developer to specialize the isolation strategy at compilation or deployment time. It supports Intel MPK, EPT, and other isolation mechanisms. PKRU-Safe [37] can provide isolation for applications mixed with both safe language and unsafe language and reduce the impact of memory-corruption vulnerabilities.

Hardware Extensions for Isolation: Researchers also introduce hardware extensions to provide high-performance isolation [22–24, 59, 65, 73–75]. Section 2.3 has introduced Mondrix [73] and CODOMs [65]. IMIX [24] extends the x86 ISA with dedicated instructions to access protected memory regions and can protect data for applications. CHERI [74] and its following works extend the MIPS and RISC-V architectures to support software compartmentalization and enforce memory safety. Donky [59] uses a software-hardware co-design for intra-process memory isolation on x86 and RISC-V platforms. XPC [22] introduces a hardware extension to speed up the switching between isolated environments. PENGLAI [23] provides a scalable memory protection for RISC-V CPU.

Call Gate (Intel): A Call Gate is a mechanism to change privilege levels and runs a predefined function using IA-32 mode CALL/JMP FAR instruction in Intel processors. It requires setting up call gate descriptors in the GDT or LDT. When such an instruction is executed, the CPU switches to a new privilege level and address according to gate descriptors. For both Call Gate and ISA-Grid, the destination level/domain and address are specified in advance. But ISA-Grid's switching mechanism validates the address of the gate instruction and uses a cache to speed up gate looking up.

Microkernel: Microkernel uses process-level isolation to deprivilege some code but faces the tradeoff of security and performance. Fine-grained isolation brings more IPCs (inter-process communications) and downgrades the system performance. Modern microkernels need hundreds of cycles even for a one-way fast path IPC [2]. Furthermore, a single exploitation in the kernel can still make all the privileged ISA resources available.

10 CONCLUSION

In this work, we propose ISA-Grid, a new architectural extension to provide fine-grained privilege control for instructions and registers, which we refer to as ISA resources. ISA-Grid allows the software to create multiple ISA domains and give different ISA access privileges to each ISA domain. Such a method could control instructions individually and control registers at bit-level. With ISA-Grid, developers can decompose software into different compartments and selectively grant them privileges to access ISA resources. We use several use cases to show the capability of ISA-Grid. We have implemented the prototypes of ISA-Grid in a RISC-V Rocket Core on an FPGA board and an x86 core on the Gem5 simulator. The evaluation shows that ISA-Grid has negligible runtime overhead (**less than 1%** for real-world applications) and acceptable hardware cost.

11 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by China National Natural Science Foundation (No.62202289, 62132014, 61925206), High-Tech Support Program from Shanghai Committee of Science and Technology (No. 22511101102), and a research grant from Huawei Technologies.

REFERENCES

[1] 2019. Ariane RISC-V CPU. https://github.com/pulp-platform/ariane.

[2] 2022. seL4 Benchmark Performance. https://sel4.systems/About/Performance/home.pml

[3] 2022. SQLite. https://www.sqlite.org/. https://www.sqlite.org/

[4] 2022. Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html. https://www.xilinx.com/products/design-tools/vivado.html

[5] Referenced Feb 2022. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest.

[6] Referenced Feb 2022. Arm Cortex-A53 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/ddi0500/j/.

[7] Referenced Feb 2022. Armv6-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0419/e/.

[8] ARM. 2022. Mbed TLS. https://tls.mbed.org/. https://tls.mbed.org/

[9] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[10] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 90–102.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.

[12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 335–348.

[13] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[14] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[15] Erik-Oliver Blass and William Robertson. 2012. TRESOR-HUNT: attacking CPU-bound encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 71–78.

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy*. 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[17] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 56–71.

[18] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1409–1426.

[19] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[20] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 81–96. https://doi.org/10.1145/2541940.2541986

[21] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. 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*. 191–206.

[22] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. 2019. XPC: Architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*. 671–684.

[23] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave.. In *OSDI*. 275–294.

[24] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX:In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)*. 83–97.

[25] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 255–267.

[26] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. {EPK}: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 609–624.

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

[28] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. A {Hardware-Software} Co-design for Efficient {Intra-Enclave} Isolation. In *31st USENIX Security Symposium (USENIX Security 22)*. 3129–3145.

[29] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 489–504.

[30] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 393–405.

[31] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. 2018. EPTI: Efficient Defence against Meltdown Attack for Unpatched {VMs}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 255–266.

[32] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security 17)*. 541–556.

[33] Zhichao Hua, Yang Yu, Jinyu Gu, Yubin Xia, Haibo Chen, and Binyu Zang. 2021. TZ-container: Protecting container from untrusted OS with ARM TrustZone. *Science China Information Sciences* 64, 9 (2021), 192101.

[34] Tyler Hunt, Zhipeng Jia, Vance Miller, Hunt Tyler, Jia Zhipeng, Miller Vance, Christopher J. Rossbach, and Emmett Witchel Witchel. 2019. Isolation and Beyond: Challenges for System Security. In *The Workshop on Hot Topics in Operating Systems (HotOS 19)*. ACM.

[35] Intel. 2022. Intel software developer's manual. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html.

[36] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*. 1445–1461.

[37] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 132–148.

[38] Dmitry Kuznetsov and Adam Morrison. 2022. Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.

[39] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1441–1454. https://doi.org/10.1145/3243734.3243748

[40] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–482.

[41] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. Twinvisor: Hardware-isolated confidential virtual machines for arm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 638–654.

[42] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 285–298.

[43] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 49–64.

[44] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1607–1619.

[45] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 115–128.

[46] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[47] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[48] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[49] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 269–284.

[50] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 116–132.

[51] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the security of arm debugging features. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 602–619.

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

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

[54] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. 2019. VoltJockey: Breaking SGX by software-controlled voltage-induced hardware faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 1–6.

[55] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (sep 1994), 1467–1471. https://doi.org/10.1145/186025.186041

[56] Philip Reames. 2021. Unintended Instructions on x86. https://github.com/preames/publicnotes/blob/master/unintended-instructions.rst.

[57] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. 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*. 546–558.

[58] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[59] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. 1677–1694.

[60] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and Operating System Support for System Call Security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 42–57. https://doi.org/10.1109/MICRO50266.2020.00017

[61] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 143–156.

[62] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.

[63] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings fo the 27th USENIX Security Symposium*. USENIX Association.

[64] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1041–1056.

[65] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with code-centric memory domains. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 469–480.

[66] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[67] Jun Wang, Xi Xiong, and Peng Liu. 2015. Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 361–373.

[68] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 592–607.

[69] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating code from data in x86 binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 522–536.

[70] Asanovic Waterman and SiFive Inc Hauser. 2021. The RISC-V instruction set manual volume II: Privileged architecture Document Version 20211203. *CS Division, EECS Department, University of California, Berkeley* (2021).

[71] Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. 2016. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro* 36, 5 (2016), 38–49. https://doi.org/10.1109/MM.2016.84

[72] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[73] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the twentieth ACM symposium on Operating systems principles*. 31–44.

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

[75] Yubin Xia, Dong Du, Zhichao Hua, Binyu Zang, Haibo Chen, and Haibing Guan. 2022. Boosting Inter-process Communication with Architectural Support. *ACM Transactions on Computer Systems (TOCS)* 39, 1-4 (2022), 1–35.

[76] Yubin Xia, Zhichao Hua, Yang Yu, Jinyu Gu, Haibo Chen, Binyu Zang, and Haibing Guan. 2021. Colony: A privileged trusted execution environment with extensibility. *IEEE Trans. Comput.* 71, 2 (2021), 479–492.

[77] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 640–656.

[78] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 203–216.

[79] Zhangkai Zhang, Yueqiang Cheng, and Zhoujun Li. 2020. Super Root: A New Stealthy Rooting Technique on ARM Devices. In *International Conference on Applied Cryptography and Network Security*. Springer, 344–363.