RegVault II: Achieving Hardware-Assisted Selective Kernel Data Randomization for Multiple Architectures

RUORONG GUO, Zhejiang University, Hangzhou, China
YANGYE ZHOU, Zhejiang University, Hangzhou, China
JINYAN XU, Zhejiang University, Hangzhou, China
WENBO SHEN, Zhejiang University, Hangzhou, China
YAJIN ZHOU, Zhejiang University, Hangzhou, China
RUI CHANG, Zhejiang University, Hangzhou, China

Memory corruption vulnerabilities pose a significant threat to system security. The traditional paging-based approach cannot protect fine-grained runtime data (e.g., function pointers), which are often mixed with other data in memory. To protect the runtime data, data space randomization is proposed to encrypt the in-memory data so that the attacker cannot control the decrypted result. Unfortunately, current hardware does not provide dedicated support for fine-grained data encryption.

This article presents RegVault II, a cross-architectural hardware-assisted lightweight data randomization scheme for OS kernels. To achieve robust, fine-grained, and lightweight data protection, we first identify five required capabilities for efficient and secure data randomization. Guided by these requirements, we design and implement novel hardware primitives that provide cryptographically strong encryption and decryption, thus ensuring both confidentiality and integrity for register-grained data. At the software level, we propose identification- and annotation-based approaches to automatically mark sensitive data and instrument the corresponding load and store operations. We also introduce new techniques to protect the interrupt context and safeguard the sensitive data spilling. We implement RegVault II on an actual FPGA hardware board for RISC-V and on QEMU for Arm, applying it to protect six types of sensitive data in the Linux kernel. Our thorough security and performance evaluations show that RegVault II effectively defends against a broad range of kernel data attacks while incurring minimal performance overhead.

CCS Concepts: • **Security and privacy** → **Systems security;**

This submission is an extended version of the article "RegVault: Hardware Assisted Selective Data Randomization for Operating System Kernels," originally published in the proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22). The added content includes the following enhancements: (1) We identify the required capabilities for robust, fine-grained, and lightweight data protection to guide the design of RegVault II. (2) We extend our design to support the Arm platform, addressing Arm-specific challenges. (3) We implement cryptographic primitives for Arm architecture and thoroughly evaluate the system, demonstrating low-performance overhead. (4) We analyze RegVault II's capabilities for defending against a broader range of attacks.
This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200).
Authors' Contact Information: Ruorong Guo, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: rrguo@zju.edu.cn; Yangye Zhou, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: zhouyangye@zju.edu.cn; Jinyan Xu, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: phantom@zju.edu.cn; Wenbo Shen (Corresponding author), Zhejiang University, Hangzhou, Zhejiang, China; e-mail: shenwenbo@zju.edu.cn; Yajin Zhou, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: yajin_zhou@zju.edu.cn; Rui Chang, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: crix1021@zju.edu.cn.

ACM 0734-2071/2025/06-ART4
https://doi.org/10.1145/3734521

Additional Key Words and Phrases: Data randomization, hardware-assisted security, RISC-V, Arm

ACM Reference Format:
Ruorong Guo, Yangye Zhou, Jinyan Xu, Wenbo Shen, Yajin Zhou, and Rui Chang. 2025. RegVault II: Achieving Hardware-Assisted Selective Kernel Data Randomization for Multiple Architectures. *ACM Trans. Comput. Syst.* 43, 1-2, Article 4 (June 2025), 34 pages. https://doi.org/10.1145/3734521

1 Introduction

Memory safety issues (e.g., memory corruption and disclosure) remain one of the biggest threats to modern systems. According to the report of Microsoft [21], around 70% of all patches were for memory corruption-related bugs. With memory corruption capability, attackers first intend to corrupt code, leading to code corruption attacks. They either modify existing code or inject new code, achieving arbitrary code execution. Luckily, memory paging can effectively defend against code corruption attacks by grouping the code into pages and enforcing the W⊕X on these pages.

Runtime data becomes a prime attacking target once code corruption is defeated, as attackers can exploit memory corruption vulnerabilities to overwrite control data for hijacking or corrupt non-control data for data-oriented attacks. At runtime, security-critical data is pervasive throughout the system, appearing in various forms across memory. These runtime data structures–encompassing function pointers, return addresses, user credentials, and other sensitive information–are frequent attack vectors. Numerous studies[1, 32, 60, 65, 71] have shown that attacks targeting runtime data are both widespread and critical. For instance, control flow hijacking techniques, such as **return-oriented programming (ROP)**, manipulate function pointers and return addresses, while data-oriented attacks, including buffer overflows and format string vulnerabilities, target non-control data like user credentials.

In the Linux kernel, security-critical runtime data is often stored adjacent to non-critical data. Furthermore, this security-sensitive data tends to be small, typically fitting within a register-sized unit, such as an 8-byte pointer on 64-bit systems. The combination of small, sensitive data residing alongside non-critical data renders traditional, page-grained protections inadequate. For instance, memory paging protections usually operate at a granularity of 4 KB, which is far too coarse to effectively isolate and safeguard small data objects like function pointers or user credentials. As a result, security mechanisms that depend solely on page-grained protection do not provide comprehensive defense against attacks targeting small-grained data. Hence, a generic and fine-grained runtime data protection mechanism is essential for safeguarding small-grained runtime data from emerging attacks.

To protect the small-grained runtime data, researchers proposed data space randomization [15, 18], which randomizes data when writing it to memory and de-randomizes it upon loading, using a set of cryptographic primitives. Although attackers can still read or overwrite the data, they cannot recover the actual value or manipulate it meaningfully without the secret keys, as any injected data becomes garbage upon de-randomization. Consequently, data space randomization has emerged as a promising approach for defending against data-oriented attacks. A key factor in data space randomization is the choice of cryptographic primitives. Due to the lack of hardware support for robust cryptographic operations, many existing schemes [15, 18, 67] rely on XOR to minimize overhead. However, these XOR-based solutions are vulnerable to memory disclosure attacks. Meanwhile, other research efforts have used AES to encrypt both control data [46] and non-control data [19, 60, 61]. Yet AES also proves inadequate for runtime data randomization. First, it imposes heavy performance overhead: AES encryption typically requires a dozen or more cycles, creating significant latency compared with a single data load instruction. Second, because AES is designed for general data encryption, it is unsuited to protect register-sized runtime data.

Beyond the cryptographic algorithm itself, the security capabilities of the chosen primitives are equally important. In response to the growing demand for small-grained runtime data protection, hardware vendors have introduced specialized features, such as Arm **Pointer Authentication** (**PA**) [66]. However, Arm PA is designed specifically for pointer integrity; it does not adequately protect data confidentiality. Consequently, deciding which cryptographic algorithm to employ and defining the necessary security capabilities to achieve robust yet efficient data space randomization remain open research challenges.

In this article, we take an initial step toward addressing these challenges by introducing RegVault II, a cross-architectural, hardware-assisted selective data randomization scheme for OS kernels. We begin by distilling five essential capabilities for efficient and secure data randomization: confidentiality and integrity, cryptographically strong encryption, flexible binding, atomicity, and register-grained support. Guided by these requirements, we design and implement novel hardware primitives that integrate the QARMA [8] algorithm, providing efficient, robust encryption at the register-grained level of runtime data.

To fully leverage these hardware primitives, we develop complementary software techniques. Specifically, we propose an auto-identification method for certain sensitive data types (e.g., function pointers) and an annotation-based approach for data that kernel developers mark as sensitive, thus ensuring both strong security guarantees and advanced flexibility. Moreover, to ensure comprehensive coverage, RegVault II introduces two new techniques–chain-based interrupt context protection to safeguard the interrupt context and cross-call spilling protection to protect spilled registers. Building on these techniques, RegVault II extends the LLVM compiler to automatically instrument stores and loads of sensitive data with encryption and decryption primitives.

To demonstrate the effectiveness of RegVault II, we develop prototypes on a real FPGA hardware board for RISC-V and QEMU for Arm. Our implementation protects runtime data in the latest Linux kernel v6.6.2, covering two types of control data and four types of non-control data. We conduct comprehensive evaluations to assess both security and performance. The security analysis confirms that RegVault II effectively defends against kernel data attacks, while performance evaluations reveal minimal overhead, at most 6.0% for micro-benchmarks and nearly zero for macro-benchmarks on RISC-V, compared to 5.4% for micro-benchmarks, 4.3% for macro-benchmarks, and 3.4% for application-benchmarks on Arm.

To summarize, this article makes the following contributions.

- **Protection Capabilities**. We identify five capabilities that each data space randomization primitive should possess to achieve robust, fine-grained, and lightweight critical kernel data protection.
- **Novel Hardware Primitives**. We design and implement novel cross-architectural hardware primitives guided by our protection capabilities to ensure both the confidentiality and integrity of register-grained data. Furthermore, we propose a cryptographic look-aside buffer to minimize performance overhead further.
- **New Protection Techniques**. We propose *chain-based interrupt context protection*, which randomizes the interrupt context while ensuring its integrity, and *cross-call spill protection*, which randomizes all sensitive register data across function calls.
- **Kernel Randomization Prototype**. We implement RegVault II prototypes on a real FPGA board for the RISC-V version and on QEMU for the Arm version, leveraging our hardware primitives to protect two types of control data and four types of non-control data in Linux kernel v6.6.2.

- **Thorough Evaluation**. We conduct thorough evaluations of RegVault II. Our security evaluation shows that RegVault II effectively defends against kernel data attacks. The performance overhead on the RISC-V version is 3.7% for UnixBench, 6.0% for LMbench, and nearly zero for SPEC2017, while the Arm version incurs overheads of 4.2% for UnixBench, 5.4% for LMbench, 4.3% for SPEC2017 and 3.4% for applications.
- **Community Contributions**. We plan to open source both the hardware and software implementation of RegVault II at `https://github.com/`. We believe hardware primitives and the software prototype can benefit the RISC-V, Arm, and Linux kernel communities.

The remainder of this article is organized as follows: We provide the background knowledge of RISC-V, Arm, and tweakable block ciphers in Section 2. Our threat model and the required capabilities are described in Section 3. We then introduce RegVault II hardware design in Section 4, including the randomization primitives and the cryptographic engine. Further, we present kernel data randomization based on our cryptographic primitives in Section 5. We give evaluations from both security and performance perspectives in Section 6. Related works are discussed in Section 7. Finally, we conclude the whole article in Section 8.

2 Background

The design and implementation of hardware primitives depend highly on the underlying hardware platform. This section introduces the necessary background for our work, covering the widely used RISC-V and Arm architectures and tweakable block ciphers.

2.1 RISC-V Architecture

2.1.1 RISC-V ISA. RISC-V is a **reduced instruction set computer** (**RISC**) **architecture** (**ISA**). It comprises three standard privilege modes: machine mode, supervisor mode, and user mode. Machine mode is usually used to manage hardware platforms and secure execution environments on RISC-V. The supervisor mode is intended to run the operating system. And the user mode is used for applications [77].

The RISC-V ISA employs **control and status registers** (**CSRs**) for system support, debugging, tracing, and performance monitoring. It reserves a 12-bit encoding space for up to 4096 CSRs, with each register being 64-bit wide in a 64-bit architecture. Additionally, a range of CSR encodings is allocated for custom registers designed by developers. CSRs tied to specific privilege levels are accessible only at those levels or higher, and any access violation triggers an illegal instruction exception.

2.1.2 Rocket Core. Rocket core is an open-source RISC-V processor implementation generated by the Rocket Chip Generator [7]. Built upon Scala and Chisel HDL [12], the Rocket core is a highly configurable and extensible in-order processor capable of running Linux. The Rocket pipeline consists of 5 stages: **instruction fetch** (**IF**), **instruction decode** (**ID**), **execute** (**EX**), **memory access** (**MEM**), and b (**WB**). Every instruction takes at least one cycle at each stage before committing. Rocket uses the **Simple Custom Instruction Extension** (**SCIE**) as an interface for custom instruction extensions. It embeds the customized functional unit into the pipeline. In addition, Rocket also introduces the **Rocket Custom Coprocessor** (**RoCC**) interface for custom instruction extensions. RoCC hides the complex implementation details of Rocket and can be used easily to communicate with the processor core, memory, and floating-point unit through a decoupled interface. The modular design decouples the RoCC and the pipeline, allowing developers to focus their efforts on the implementation of the functionality. For this reason, most of the previous works [24, 25, 34, 45] are based on the RoCC interface. We also implement RegVault II extensions with Rocket Core through this interface.

2.2 Arm Architecture

Arm is also a RISC architecture for computer processors. Arm architecture is known for its power-efficient design, making it a popular choice for mobile and embedded systems. Arm processors operate under different privilege levels: EL0 for the user, EL1 for the kernel, EL2 for the hypervisor, and EL3 for the security monitor.

In recent years, Arm architecture introduced PA, a security feature to mitigate certain types of software attacks, such as ROP. PA uses a cryptographic technique to sign and authenticate pointers, ensuring that they have not been tampered with. This adds a layer of security by verifying the integrity of pointers before their use, thereby protecting against unauthorized changes and enhancing the system's overall security.

2.3 Tweakable Block Ciphers

A tweakable block cipher [42] refines the traditional two-input design of a block cipher by introducing a tweak as an additional input, enhancing the variability of the ciphertext. It operates on three inputs: a key, a tweak, and a plaintext, generating a single ciphertext as output. By modifying the tweak, the cipher can produce different ciphertexts even when using the same key and plaintext, thereby increasing encryption flexibility. This tweak mechanism enhances randomness and security, making it more resistant to cryptanalysis and replay attacks–a function similar to the role of the **initialization vector** (**IV**) in the AES algorithm. Any public information can be used as a tweak, and the overhead of modifying a tweak is much lower than regenerating a key. Nevertheless, the protection of a tweakable block cipher is still dependent on its key. It guarantees that the secret key will not be disclosed even if the attacker controls the tweak.

QARMA block cipher family. QARMA [8] is introduced as a family of lightweight tweakable block ciphers targeted at applications such as software protection and memory encryption. It is a three-round Even-Mansour construction. One cryptographic operation contains several forward rounds, a pseudo-reflector, and several backward rounds. QARMA has two variants, supporting block sizes of $n = 64$ bits and $n = 128$ bits. Input message, output, and tweak are all n bits long, and the key is always $2n$ bits long. Furthermore, r determines the actual rounds of forward and backward operations. The encryption algorithm QARMAr has r rounds for forward and another r for backward.

The design of QARMA takes both security and hardware implementation into account. It is claimed that even when r is relatively small, such as 5, QARMA shows the capability of resisting common attacks on block ciphers, algebraic attacks [22], and invariant subspace cryptanalysis [39]. Its low hardware latency and minimal area overhead make it a seamless fit within the processor.

3 Threat Model and Requirements

3.1 Threat Model and Assumptions

In our threat model, the attacker takes full control over the user space, allowing them to run programs and invoke system calls. Additionally, the attacker can exploit kernel vulnerabilities to achieve arbitrary kernel memory read and write capabilities, but these reads must go through memory, and the attacker cannot directly read the registers. Given these capabilities, the attacker can overwrite or substitute any randomized data in kernel memory, including all control data (such as return addresses and function pointers) and certain security-critical types of non-control data (such as user credentials). In the meantime, we assume the kernel text section in memory is protected by existing techniques[50] so that it is not affected by the vulnerabilities. Side-channel attacks are beyond the scope of this article, consistent with assumptions made in existing selective data protection research. We also assume that the hardware operates correctly, preventing the attacker from exploiting hardware malfunctions to de-randomize the protected data.

3.2 Protection Capabilities

We identify five **capabilities** (**CAPs**) that data randomization hardware primitives must possess to achieve robust, fine-grained, and lightweight kernel data protection. Specifically, RegVault II aims at safeguarding the confidentiality and integrity of critical data comprehensively. Its randomization algorithm should be cryptographically robust, allowing for flexible binding methods. Furthermore, to achieve fine-grained and lightweight protection, RegVault II needs to employ register-grained primitives that operate atomically. These capabilities not only enhance security but also ensure efficiency.

Cap 1 (Confidentiality and Integrity). *To mitigate the risks imposed by arbitrary memory read and write vulnerabilities, data randomization operations must ensure both the confidentiality and integrity of critical data.*

Attacks frequently exploit arbitrary memory vulnerabilities to extract sensitive information, such as code pointer values, thereby undermining **Address Space Layout Randomization** (**ASLR**). To address these risks, we prioritize the protection of data confidentiality. To mitigate these risks, RegVault II prioritizes the protection of data confidentiality. Moreover, attackers may also leverage arbitrary memory writes to tamper with randomized data. Consequently, preserving data integrity is essential for the timely detection and mitigation of any such tampering attempts.

Cap 2 (Cryptographically Strong Algorithm). *The randomization algorithm must be cryptographically robust, ensuring resilience against adversarial cryptanalytic attacks.*

A strong cryptographic foundation guarantees that even if an attacker gains access to both the randomized data (ciphertext) and the de-randomized data (plaintext), they cannot infer the randomization keys. The confidentiality of randomization keys is crucial, as any compromise could jeopardize the entire security model. By employing a strong cryptographic algorithm, RegVault II ensures that randomization keys remain secure, resistant to inference, and protected from cryptanalytic attacks.

Cap 3 (Flexible Binding). *To defend against substitution attacks, the data randomization operations must support flexible binding, allowing the same data to be randomized into different representations based on its usage context.*

Substitution attacks replace randomized data with another chosen randomized data, potentially bypassing security mechanisms. To mitigate this risk, RegVault II ensures that randomized data can adapt dynamically depending on the execution context. This context-aware randomization strengthens system security by preventing attackers from substituting and replaying pre-randomized values.

Cap 4 (Atomic Operation). *The data randomization operations must be atomic to achieve lightweight protection and eliminate intermediate result leakages.*

Existing cryptographic extensions, such as RISC-V Cryptography Extension [20] and Intel AES-NI extension [2], provide round encryption instructions to speed up the encryption. Unfortunately, the user still needs multiple rounds of instructions to perform a complete AES operation, which might leak the round key and intermediate results in memory. The randomization operation should require as few instructions as possible to support lightweight usages. To eliminate intermediate result leakages, the randomization operations are required to be atomic.

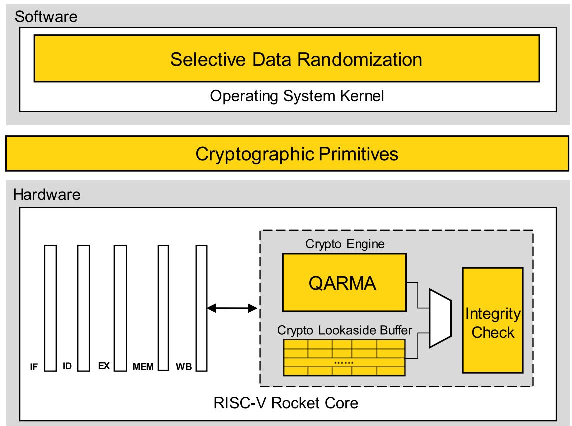


Fig. 1. Overview of RegVault II.

Cap 5 (Register-grained Randomization). *Kernel data structures often operate at the register level, necessitating a register-grained randomization approach for effective protection.*

Most existing security mechanisms rely on page-grained memory protection, which adversaries can bypass by targeting smaller, register-sized data such as function pointers, return addresses, and other critical non-control data. To counter these evolving attack strategies, RegVault II needs to support register-grained randomization, providing fine-grained security that enhances the resilience of the kernel against sophisticated attacks that exploit coarse-grained memory protections.

In the following sections, we first discuss the design of RegVault II hardware primitives and highlight their support for the above requirements Section 4. Then, we further discuss adopting these hardware primitives to achieve effective kernel data protection Section 5.

4 RegVault II Hardware Architecture

4.1 Overview

RegVault II is a hardware-assisted selective data randomization architecture for operating system kernels. The basic idea of RegVault II is to encrypt the specified data before storing it in memory and decrypt it after loading it into registers. As shown in Figure 1, RegVault II consists of both the hardware and the software support. For the hardware, RegVault II extends the 64-bit ISAs to implement the lightweight cryptographic primitives, which are used to encrypt and decrypt the selected data to achieve selective data randomization (Section 4.2). RegVault II involves extending the instruction sets of RISC-V and Arm, designing corresponding cryptographic instruction encoding formats based on the characteristics of their respective instruction sets (Sections 4.3 and 4.4). To support these cryptographic primitives, RegVault II designs and implements a hardware-level crypto-engine. The crypto-engine includes a look-aside buffer to further reduce the performance overhead (Section 4.5). The specific implementation of cryptographic instructions on the RISC-V and Arm platforms is stated in Section 4.6. To demonstrate the effectiveness of the proposed techniques, we apply RegVault II to protect the data in the kernel, which will be detailed in Section 5.

4.2 Common Primitive Design

RegVault II chooses to encrypt the selected data for data randomization. As a result, RegVault II needs to decrypt this data when using it. The cryptographic primitives should be lightweight, as the protected data may be used frequently. Moreover, the cryptographic primitives also need to fulfill all requirements mentioned in Section 3.2. Therefore, in the following, we discuss how RegVault II primitives fulfill these needs in both algorithm selection and instruction design.

4.2.1 Encryption Algorithm Selection. XOR [14, 15] is a general and straightforward encryption algorithm. However, with the plaintext and the ciphertext, the attacker can easily infer the XOR keys so that they can steal and tamper with confidential data. Therefore, XOR is not cryptographically strong and cannot provide CAP 1 and CAP 2. Moreover, there are works on using AES to encrypt control data [46] or to achieve selective data protection [19, 60, 61]. AES has already been implemented on the hardware, so people will naturally use it for data protection. Nevertheless, AES is not designed to protect runtime data. First, AES is too heavy for runtime data randomization. AES encryption usually requires about a dozen processor cycles based on hardware implementation or a dozen instructions based on software. Compared with the single data loading instructions, the AES operation introduces prohibitive performance overhead. Second, AES does not support flexible binding and is usually not atomic either, thus failing to provide CAP 3 and CAP 4. Third, AES is designed for general data encryption, which is unsuitable for runtime data encryption, especially for register-size data protection, such as return addresses, function pointers, or other sensitive data in the kernel. So it is not a proper algorithm for CAP 5.

Instead of using XOR and AES, RegVault II chooses the *QARMA* [8], a lightweight tweakable block cipher, to encrypt the sensitive data. The tweakable block cipher is tailored for resource-constrained devices [48]. It also adopts a hardware-implementation-friendly design with lower latency, smaller area, and lower power consumption. In addition to the secret encryption keys, the tweakable block cipher provides an additional input, called *tweak*, providing more diversity to the encryption. In particular, a small tweak change leads to a different encryption result, even with the same plaintext. Moreover, the tweak does not need to be kept secret. Even with the tweak, there is no way for the attacker to infer the ciphertext or the encryption keys. Both the tweak in QARMA and the IV in AES introduce variability into the encryption process, yet they serve distinct functions. The AES IV is a typically random value used to prevent ciphertext repetition for identical plaintexts under the same key, ensuring semantic security. In contrast, the tweak in a tweakable encryption scheme is an application-specific parameter that provides deterministic variability without necessitating randomness. It enables context-dependent encryption by ensuring that identical plaintext segments are processed differently within the same cryptographic session. This distinction allows tweaks to support fine-grained security policies in scenarios such as disk encryption, where data blocks are encrypted uniquely despite sharing the same key.

In our design, RegVault II chooses the QARMA cipher with a block size of 64 bits, which matches the register size of a 64-bit architecture. RegVault II also chooses seven rounds and the sbox σ_2 parameter for the strongest data confidentiality to defend against cryptanalysis attacks. QARMA is a cryptographically strong algorithm, which provides CAP 2. Based on QARMA, RegVault II implements lightweight cryptographic protection primitives, which accept the data context as the tweak to achieve the flexible binding, as required in CAP 3.

4.2.2 Cryptographic Instruction Design. To support kernel data randomization, RegVault II extends both RISC-V and Arm instruction sets to implement the QARMA-based cryptographic operations.

Instruction atomicity: To achieve atomicity, RegVault II implements the QARMA cryptographic operations in a single encryption instruction and a single decryption instruction. Therefore, the encryption and decryption are atomic in RegVault II, preventing the intermediate result leaks, which fulfill the requirement of CAP 4. In this way, RegVault II hides the details of the round keys from the

Table 1. RegVault II Cryptographic Instruction Extension

Arm Privilege Level	RISC-V Privilege level	Master key	General key
EL0	User	-	-
EL1	Supervisor	X	WX
EL3	Machine	RWX	RWX

(a) Read(R), Write(W), and Execute(X) permission of RegVault II cryptographic key registers for different privilege levels.

Instruction	Description
cremk rd, {rs,} tweak, start, end	Encryption with master key
crdmk rd, {rs,} tweak, start, end	Decryption and check integrity with master key
cre[a-g,t]k rd, {rs,} tweak, start, end	Encryption with general key [a-g,t]
crd[a-g,t]k rd, {rs,} tweak, start, end	Decryption and check integrity with general key [a-g,t]

(b) RegVault II cryptographic instruction description, RISC-V version has both destination (`rd`), and source (`rs`), while Arm version has only one destination register.

software, delegating the generation and use of the round keys to hardware. Software only needs to provide the tweak and the data to be randomized, and then wait for the final randomization result.

Operand granularity: The cryptographic instruction processes a single register operand at a time. As a result, RegVault II is capable of offering register-grained protection for runtime data, effectively addressing the requirement for CAP 5. When only specific bytes in a register need protection, RegVault II supports range selection to encrypt these targeted bytes. To ensure data integrity, it fills the unselected bytes with `0xff` before encrypting the entire register. After decryption, RegVault II checks if these bytes remain `0xff`. If any alteration is detected, the entire data is considered compromised, and an exception is raised. By leveraging this register-grained randomization, RegVault II provides a more precise and effective defense against targeted attacks that exploit vulnerabilities in runtime data, which are often overlooked by coarser protection mechanisms such as page-grained randomization.

Key management: RegVault II provides dedicated key registers to hold the secret keys by extending the CSRs in RISC-V and system registers in Arm. The key register is 64-bit, but the key used in QARMA-64 is 128-bit. Therefore, we divide a key into the high and low parts and save them in 2 key registers. We can reuse the instructions in the standard instruction set to read and write these key registers.

RegVault II supports two kinds of encryption keys: master key and general key. The master key is designed to generate only the general keys. For security considerations, it should not be used to encrypt data directly. The general keys are designed for data encryption. The access permission of cryptographic keys is shown in Table 1(a). Both keys permit reading, writing, and executing (i.e., used by the cryptographic instructions) in machine mode. In supervisor mode, the kernel can only execute instructions using the master key. However, no direct reading or writing is allowed. By contrast, the general keys can be written and executed. In user mode, the application cannot access any key registers.

4.3 RISC-V Primitive Design

On RISC-V architecture, the customized cryptographic instruction set of RegVault II follows the R-type instruction format, which specifies two source registers and one destination register. Moreover, we need one additional key register to support the QARMA cryptographic operations. Therefore, the extended RISC-V instruction accepts three inputs: the plaintext (or the ciphertext)

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
end	start	mod	rs2	rs1	funct3	rd	opcode
3	3	1	5	5	3	5	7
		enc/dec	tweak	src	key selection	dest	RegVault

Fig. 2. RegVault II RISC-V cryptographic instruction format.

in `rs1`, the tweak in `rs2`, and the cryptographic key selected by `funct3`. Figure 2 shows the instruction format. The extended instructions are 32-bit long. Bits [31:26] specify the selected range of 8 bytes in the register, in which bits [31:29] and [28:26] index the end and start byte, respectively. To protect data integrity, we fill 1 to the unselected bytes, detailed in Section 5.5.1. Bit 25 specifies whether the operation is encryption or decryption. Bits [24:20] and [19:15] index the general-purpose registers for the tweak and the plaintext (or the ciphertext), respectively. Bits [14:12] index the key registers. The current format supports eight key registers: 1 master key register and seven general key registers. Bits [11:7] are used to index the general-purpose register for the output. Bits [6:0] hold the opcode.

Table 1(b) lists the detailed instructions. The `cre[x]k` (context-aware register encryption) instruction encrypts the given value in `rs` with the tweak in `tweak` and the key in key register `x` and puts the result in register `rd`. The key selected by bits [14:12] is directly loaded from the key register and used by the customized instruction to perform the cryptographic operation. The `crd[x]k` instruction follows a similar format but is for decryption. Referring to key registers, the `x` can either be the master key `m` or the general key `a-g`. Since our goal is to randomize the kernel data, the execution of the extended instructions is restricted to supervisor mode. However, RegVault II is designed for general data randomization; it can be easily extended to the user mode with userspace-related CSR extensions.

4.4 Arm Primitive Design

Adapting RegVault II to the Arm platform faces several technical challenges due to both hardware constraints and a more complex software ecosystem compared to the RISC-V version. At the hardware level, Arm instructions have fewer bits available to customize the required registers and range selection functionality. To address this, we first reduced the register requirement by using the same register for both the source and destination. Second, instead of allowing full flexibility in selecting arbitrary start and end bytes as in the RISC-V design, we introduce a direction bit that fixes the start byte to either the 0th or the 7th position while still permitting flexible selection of the end byte.

At the software level, the Arm toolchain integrated within QEMU proved to be far more complex than the RISC-V ecosystem that uses the Spike emulator and a straightforward LLVM compiler. This increased complexity required substantial modifications across multiple layers, including QEMU's abstraction, architecture-specific parts of the LLVM compiler, and even the Linux kernel. These extra efforts are necessary to fully support the advanced features of Arm and to meet the broader security challenges encountered in real-world systems.

Despite these challenges, we successfully designed and implemented data space randomization primitives on the Arm architecture with comprehensive software toolchain support. Given that Arm is more widely used in real-world computing products and faces more diverse security challenges than RISC-V, supporting memory security mechanisms on Arm has a greater practical impact.

Arm instruction formats: Since Arm instruction formats occupy more bits than RISC-V, fewer bits are available for customization. To conserve enough bits for other fields, the customized cryptographic instruction set of RegVault II Arm uses the data-processing (1 source) instruction

31	30	29	28 21	20	19	18 16	15 12	12 10	9 5	4 0
sf	1	S	11010110	mod	dir	range	opcode	keysel	rn	rd
1	1	1	8	1	1	3	3	3	5	5
				enc/dec				key selection	tweak	src/dest

Fig. 3. RegVault II Arm cryptographic instruction format.

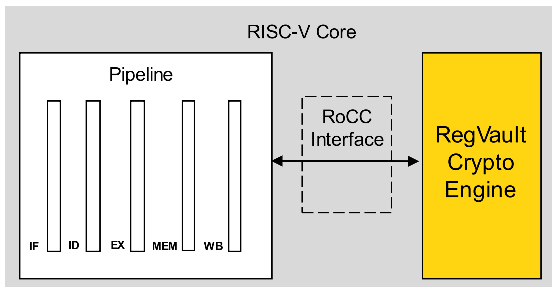


Fig. 4. The crypto-engine is integrated into the Rocket processor via the RoCC interface.

format, which specifies one source register and one destination register. Therefore, we must treat the source register as a tweak and the destination register as input and output simultaneously. The extended Arm instruction set accepts three inputs: the plaintext (or the ciphertext) in `rd`, the cryptographic key selected by bits [12:10], and the tweak in `rn`. Figure 3 shows the instruction format. The extended instruction set is 32-bit. Arm reserves the top 3 bits. Bits [28:21] are used to identify the class of the instruction. Bit 20 specifies the operation as either encryption or decryption. Since we have fewer bits to use in the Arm instruction than in RISC-V, bit 19 is chosen to indicate that we want to select data from its highest or lowest byte, and bits [18:16] to encode the length of the range. Bytes outside the range will be filled with `0xff`s for integrity check. Bits[15:13] mark this as a RegVault II instruction, and bits [12:10] index the key registers. The current format supports eight key registers: 1 master key register and seven general key registers. Bits [9:5] and bits [4:0] are used to index the general-purpose registers for the tweak and the plaintext (or the ciphertext), respectively.

The `cre[x]k` instruction encrypts the given value in `rd` with the tweak in `tweak` and the key in key register `x` and puts the result in register `rd`. The key is selected by bits [12:10] and is used in the same way as the RISC-V version. The `crd[x]k` instruction follows a similar format but is for decryption. The key registers' representation and usage are the same as the RISC-V version.

4.5 Crypto-engine Design

RegVault II implements a crypto-engine to perform the actual cryptographic operations to support the hardware primitives. As shown in Figure 4, the crypto-engine on the RISC-V platform is integrated into the Rocket processor core via the RoCC interface. More specifically, when a cryptographic instruction is executed, the engine first checks whether the current privilege level is either supervisor mode or machine mode. Next, it performs the cryptographic operation and generates the result in the destination register. For data randomization, encryption and decryption instructions are invoked frequently. Therefore, the number of cycles consumed on the cryptographic operation directly determines the RegVault II performance. To reduce the cycles, we introduce a cache to buffer the recently used cryptographic results (Section 4.5.1).

Figure 5 gives the overview of the RegVault II crypto-engine. The crypto-engine consists of three components—the **control unit** (**CU**), the **arithmetic unit** (**AU**), and the **cryptographic lookaside buffer** (**CLB**). The CU drives the other components according to the input signals. It checks the privilege, selects the key, and sends the encrypt or decrypt signal to AU (for calculation).

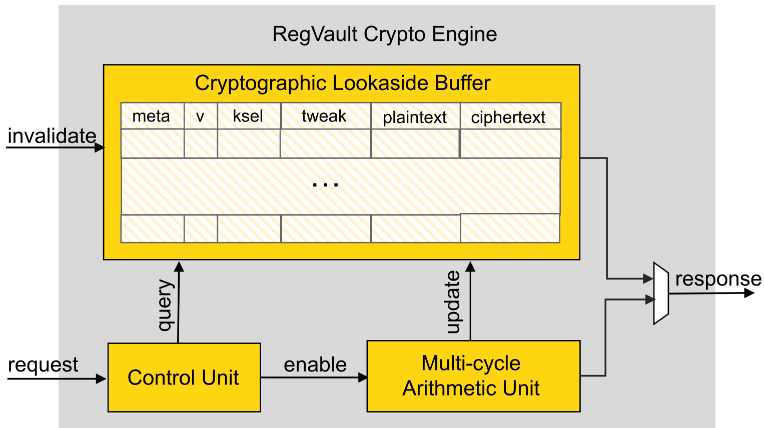


Fig. 5. The overview of RegVault II crypto-engine structure with cryptographic lookaside buffer for acceleration.

The AU accepts the input and does the actual encryption or decryption operations. The CLB is designed to buffer the result for performance optimization. The design and implementation of CU and AU are straightforward; thus, we skip the details. In the following subsections, we present the details of the CLB.

4.5.1 Cryptographic Lookaside Buffer. Compared to normal operations, cryptographic operations, such as encryption and decryption, are slow. To substantially reduce the overhead of frequently invoked cryptographic operations, we propose to use a buffer, named CLB, to hold recently calculated encrypted and decrypted results. As shown in Figure 5, being embedded in the crypto-engine, CLB can work closely with AU.

CLB stores the encrypted or decrypted results whenever AU completes a new cryptographic operation. When new cryptographic operation queries come, RegVault II crypto-engine first looks up CLB for the results before spending multiple cycles performing the cryptographic operations in AU.

CLB structure: As shown in Figure 5, the CLB consists of a configurable number of entries. Each entry of the CLB contains six elements: a timestamp for replacement `meta`, a valid bit `v`, a key selection index `ksel`, the tweak, the plaintext, and the ciphertext. The `1` value of the valid bit marks a valid entry, while `0` marks an invalid one.

CLB invalidate: To save space, CLB uses 3-bit key selection indices to replace the 128-bit keys. When different threads are running, the same key register may hold different keys. Therefore, when a key register is overwritten, the corresponding CLB entries are out of date. Therefore, whenever a key register is updated, the crypto engine invalidates the entries with the same key selection index. In RegVault II hardware implementation, the `invalidate` signal is directly sent to the crypto-engine from CSR update operations.

CLB query: To improve the hit ratio and make full use of the buffered cryptographic results, the CLB entries are fully associative. Querying the CLB involves checking all entries. If a valid entry matches the cryptographic request (i.e., matching the key, the tweak, and the plaintext for encryption), the result will be given to the pipeline in just one cycle.

CLB update: The newly generated cryptographic result should replace the old one. For simplicity, we adopt the well-known **least-recently-used** (**LRU**) cache replacement policy. To implement it,

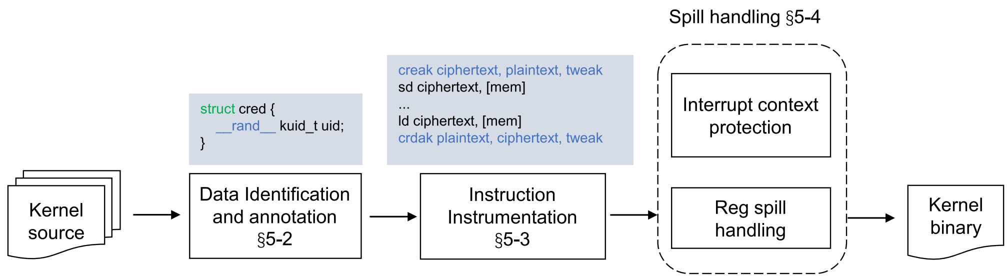


Fig. 6. RegVault II kernel data randomization process.

we use the `meta` field in CLB entries as timestamps to find the LRU entry and replace it with the new result.

Performance gain: As detailed in Section 6.2.1 and Figure 11, CLB can achieve a 50% hit ratio with 8 entries and 80% with 64 entries on UnixBench. It also achieves a 55% hit ratio with 8 entries and 70% with 64 entries on a real user-mode application, Docker. Although more CLB entries can improve the hit ratio, the matching for a large, fully associative CLB incurs a long latency, which is unacceptable in the pipeline stage. To balance the hit ratio and the hardware design, we use the CLB with eight entries to build our RegVault II prototype.

4.6 Implementation

RISC-V hardware implementation. We implement the hardware prototype of RegVault II RISC-V on the Rocket core with ~1100 lines of addition and ~200 lines of deletion in Chisel. We implement the QARMA-based crypto-engine and the decoding unit for cryptographic instructions. Moreover, we also extend the CSR registers to store master and general keys. As mentioned before, we integrate the crypto-engine with the RISC-V core via both the SCIE and RoCC interface to provide better extensibility. We use Vivado 2020.2 to check out the area overhead. The synthesized RTL module covers the Rocket processor and L1 cache. The result indicates that RegVault II with 8 CLB entries incurs a 2.9% LUT and 4.9% Reg overhead to provide cryptographic primitives.

Arm implementation: We implemented a prototype of RegVault II for Arm within the QEMU emulation environment by adding ~600 lines of C code. The code integrates the QARMA algorithm and also emulates RegVault II cryptographic instructions. The implementation ensures seamless integration with QEMU's existing CPU model. It also incorporates logging mechanisms to facilitate debugging and performance evaluation. This prototype not only demonstrates the feasibility of deploying RegVault II on Arm but also lays a solid foundation for further enhancements and potential real-world applications in secure systems design.

5 RegVault II Kernel Data Randomization

5.1 Overview

Figure 6 shows the overall flow of RegVault II kernel data randomization. With the kernel source code as the input, RegVault II first annotates the data types that need to be randomized (Section 5.2). After that, RegVault II instruments the randomized data store and loads with encryption and decryption primitives (Section 5.3). Finally, to defeat data leak and corruption from spilling, RegVault II protects both the interrupt context and register spilling (Section 5.4). The following sections describe these steps in detail.

```c
1  struct cred {
2      ...
3      kuid_t uid __rand__;
4      ...
5  };
6
7  long __sys_setuid(uid_t uid) {
8      struct cred *new = prepare_creds();
9      new->uid = kuid;
10     ...
11 }
12
13 SYSCALL_DEFINE3(setpriority, int, which, ...){
14     const struct cred *cred = current_cred();
15     ...
16     if (!uid_eq(uid, cred->uid))
17     ...
18 }
```

(a) C code of cred->uid store and load.

```asm
# __sys_setuid
# s1 holds kuid
addi  a1, s0, 8  # calculate the address of new->uid
creak a1, s1, a1, 0, 7 # encrypt kuid using storage address
sd    a1, 8(s0)  # store encrypted kuid

# sys_setpriority
ld    a1, 0(s5)  # load encrypted cred->uid
crdak a1, a1, s5, 0, 7 #decrypt cred->uid
```

(b) RISC-V assembly of cred->uid store and load. Highlighted lines are the instrumented instructions.

```asm
// __sys_setuid
// x1 holds kuid
add  x2, fp, 8  // calculate the address of new->uid
creak x1, x2, 0, 7 // encrypt kuid using storage address
str    x1, [fp, 8]  // store encrypted kuid

// sys_setpriority
ldr    x1, [x5]  // load encrypted cred->uid
crdak x1, x5, 0, 7 // decrypt cred->uid
```

(c) Arm assembly of cred->uid store and load. Highlighted lines are the instrumented instructions.

Fig. 7. Data randomization instrumentation in RegVault II.

5.2 Data Identification and Annotation

Before instrumentation, RegVault II needs to know which data needs to be randomized. To achieve this, RegVault II extends the LLVM compiler to automatically identify interested types (e.g., return address, function pointer) and instrument the load and store of them with encryption and decryption instructions. RegVault II uses this approach to protect kernel return addresses and function pointers. Besides the automated approach, RegVault II provides a macro called `__rand__` for kernel developers or RegVault II users to annotate the interested data types. Taking the `cred.uid` in Figure 7(a) as an example, to protect `uid` member, the user annotates it by putting the `__rand__` behind, as shown in Line 3. The `__rand__` adds a compiler attribute that will be recognized by RegVault II during the compiling stage. Moreover, the `__rand__` macro is a field-sensitive annotation on types rather than a single object instance. In other words, with the annotation, the `uid` members of all `cred` instances are marked for randomization.

5.3 Instruction Instrumentation

RegVault II first collects the marked data types by automatically identifying manual annotations. After that, RegVault II traverses all IR instructions to identify all loads and stores operating on the

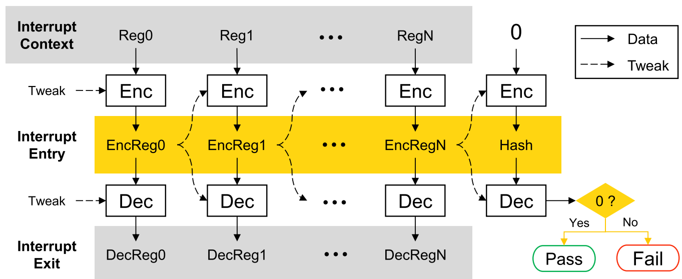


Fig. 8. RegVault II interrupt context randomization.

marked data types. Finally, RegVault II inserts data encryption instructions with proper tweaks before the data store and data decryption instructions after the data load.

Let us illustrate the instrumentation using the same example in Figure 7. For the uid store, Line 9 of Figure 7(a) assigns the uid to a new cred, while Line 5 of Figure 7(b) and (c) shows the corresponding assembly code where the `sd` stores the uid in register `a1` into the memory. RegVault II inserts two instructions at Lines 3 and 4, where the `addi` instruction calculates the address of `new->uid`, which the `creak` encrypts the uid using the address as the tweak. On the uid load side, Line 16 of Figure 7(a) loads the `cred->uid` for comparison, where the corresponding assembly code is in Line 8 of Figure 7(b) and (c). As the data is encrypted in the memory, RegVault II inserts the decryption instruction `crdak` to de-randomize. The storage address is used as the tweak for decryption, the same as encryption. Besides, to initialize annotated data allocated statically, RegVault II collects their information, generates initialization functions, and invokes these functions at kernel boot time.

Moreover, the cryptographic primitives of RegVault II are capable of randomizing 64-bit data. For data types that are smaller than 64-bit, RegVault II extends the unused top bits for integrity checking. Note that RegVault II uses the field-sensitive data type annotation to identify data loads/stores and thus does not need to conduct time-consuming points-to analysis on kernel objects [82].

5.4 Spill Handling

To guarantee that the protected data is always randomized in memory, RegVault II needs to handle two types of spilling—the interrupt context spilling and the register spilling.

5.4.1 Interrupt Context. When an interrupt happens, the kernel stores all general-purpose registers (called the interrupt context) in memory. Unprotected interrupt context gives attackers chances to leak and manipulate these register values [11], which imposes a severe threat to RegVault II. Therefore, RegVault II proposes a novel chain randomization technique, named *CIP* (short for *chain-based interrupt context protection*), which not only encrypts the interrupt context in the memory but also provides integrity protection. The basic ideas of CIP are (1) enforcing a chain-based de-randomization process so that the next register value is decrypted using the previous register value as the tweak. As a result, if the attacker corrupts one value in the middle, all the subsequent register values will be decrypted into garbage values. (2) Inserting a zero value at the end for encryption and decryption. The decrypted zero value can be used for integrity checking. In this way, CIP can detect any corrupted register values in the middle of the interrupt context.

Figure 8 illustrates the process of CIP. When an interrupt happens during kernel execution, the ith register is encrypted using the $(i-1)$th register value (Equation (1)). Therefore, when reloading the kernel interrupt context, the ith register is decrypted using the decrypted value of the $(i-1)$th register (Equation (2)), enforcing the chained decryption scheme.

$$EncReg_i = \begin{cases} Enc_K(Reg_i,\ tweak), & i = 0 \\ Enc_K(Reg_i,\ EncReg_{i-1}), & i > 0 \end{cases}, \tag{1}$$

$$DecReg_i = \begin{cases} Dec_K(EncReg_i,\ tweak), & i = 0 \\ Dec_K(EncReg_i,\ EncReg_{i-1}), & i > 0 \end{cases}. \tag{2}$$

Besides, to defeat any spatial substitution attack, CIP encrypts the first register using its storing address as the tweak. To defeat cross-data-type substitution attacks, CIP uses a dedicated key register (called the interrupt key) for interrupt context randomization, which is different from the keys for randomizing control data or non-control data. Moreover, CIP ensures that each thread maintains its own interrupt key, thwarting cross-thread substitution attacks.

5.4.2 Sensitive Register Spilling Protection. Sensitive data and their intermediate results remain as plaintext in registers and might be spilled to memory temporarily due to (1) the lack of physical registers and (2) function calls. Therefore, to achieve comprehensive protection, RegVault II must handle these spills securely.

Identifying sensitive registers: RegVault II has to identify sensitive register spills in the compiler backend, where the information, such as type and annotation, is lost. Observing that annotated sensitive data must be decrypted or encrypted whenever it enters or leaves registers, RegVault II proposes a sensitive register identification algorithm:

— For RegVault II cryptographic operations, RegVault II marks the plaintext operand as a sensitive register.
— For assignment operations, if one of the operands is sensitive, RegVault II marks the other one as a sensitive register.
— For arithmetic operations, if one of the source operands is sensitive, RegVault II marks the destination operand as a sensitive register.

Function-inner spilling protection: When running out of physical registers, the compiler will spill sensitive registers to memory. To reduce such spilling, RegVault II increases the spilling cost of sensitive registers and suggests the compiler spill other registers. If a sensitive register has to be spilled, RegVault II inserts cryptographic primitives (using stack pointer `sp` as the tweak) around the store and reload instructions.

Cross-call spilling protection: During function calls, sensitive caller-saved registers may be spilled by the caller, and sensitive callee-saved registers may be spilled by the callee (or subsequent callees). To protect these spills, RegVault II proposes a novel *CSP* (short for cross-call spilling protection). For sensitive caller-saved registers, RegVault II identifies their spilling and inserts cryptographic primitives to protect them. For sensitive callee-saved registers at call sites, RegVault II encrypts them before entering the callee and decrypts them after returning from the callee. Nevertheless, this may introduce redundant encryption and decryption. For example, during two successive calls, a sensitive callee-saved register may be decrypted after the first call and be immediately encrypted again before the second call. To address the problem, RegVault II introduces an algorithm for eliminating redundant cryptographic operations:

(1) A decryption instruction is partially removable if all uses of its decrypted result are encryption instructions using the same tweak. Similarly, an encryption instruction is partially removable if all definitions of the value to be encrypted are decryption instructions using the same tweak.

Table 2. Protected Kernel Data in RegVault II

Randomized Data		Tweak	Instrument
Control Data	Return Addr	Stack Pointer	Type
	Function Pointer	Storage Addr	Type
Non-control Data	Cred Struct	Storage Addr	Annotation
	SELinux State	Storage Addr	Annotation
	PGD Pointer	Storage Addr	Annotation
	AES Keys	Storage Addr	Manual

```
# encrypt and store a pointer (in a0)
creak a0, a0[7:0], t1   ;encrypt pointer a0 using key reg a
sd    a0, 0(s0)         ;store the encrypted pointer
# load and decrypt a pointer
ld    a0, 0(s0)         ;load an encrypted pointer
crdak a0, a0, t1, [7:0] ;decrypt the pointer
```

(a) Pointer randomization.

```
# encrypt and store 32-bit data (in the low 4 bytes of a0)
creak a0, a0[3:0], t1   ;encrypt the 32-bit data
sd    a0, 0(s0)         ;store the encrypted data
# load, decrypt, and check 32-bit data
ld    a0, 0(s0)         ;load an encrypted data
crdak a0, a0, t1, [3:0] ;decrypt-and-check the 32-bit data
```

(b) 32-bit data randomization with integrity.

```
# encrypt and store 64-bit data (in a0)
creak a1, a0[3:0], t1   ;encrypt the low 4-byte data
creak a2, a0[7:4], t2   ;encrypt the high 4-byte data
sd    a1, 0(s0)         ;store the encrypted low 4-byte data
sd    a2, 8(s0)         ;store the encrypted high 4-byte data
# load, decrypt, and check 32-bit data
ld    a1, 0(s0)         ;load the encrypted low 4-byte data
ld    a2, 8(s0)         ;load the encrypted high 4-byte data
crdak a1, a1, t1, [3:0] ;decrypt-and-check low 4-byte data
crdak a2, a2, t2, [7:4] ;decrypt-and-check high 4-byte data
or    a0, a1, a2        ;recover the original 64-bit data
```

(c) 64-bit data randomization with integrity.

Fig. 9. Data randomization using RegVault II primitives. Highlight lines are instrumented primitives.

(2) RegVault II removes a decryption instruction if it is partially removable, and all uses of its decrypted result are partially removable encryption instructions. Similarly, RegVault II removes an encryption instruction if it is partially removable, and all definitions of the value to be encrypted are partially removable decryption instructions.

5.5 Randomized Data

RegVault II protects kernel control data and non-control data. The protected data types are summarized in Table 2. The control data includes the return addresses and function pointers. However, the Linux kernel has too many types of non-control data. Without loss of generality, we apply RegVault II to protect the user credentials, security feature states, memory management data, and encryption keys to show how RegVault II can be used. With the annotation support, RegVault II can be used to protect more kernel data easily. All of the following randomization schemes are implemented on both RegVault II RISC-V and Arm platforms.

5.5.1 General Scheme. Figure 9 shows the general scheme of data protection using RegVault II primitives.

— **Pointer.** RegVault II encrypts and decrypts all bytes (i.e., with range `7:0`) in the pointers directly (Figure 9(a)). Therefore, any corrupted pointers in memory are decrypted into garbage values, pointing to illegal addresses. RegVault II also uses this way to protect 64-bit data that does not require integrity protection.

— **32-bit data with integrity check**. RegVault II extends the 32-bit data to 64-bit and fills 0 to the upper 32-bit. To achieve this, the encryption sets the range as `[3:0]` (Figure 9(b) Line 2). In the decryption, the upper 32-bit is used for the integrity check (Line 6). Therefore, RegVault II protects both confidentiality and integrity for 32-bit data.

— **64-bit data with integrity check**. RegVault II splits 64-bit data into two 32-bit data, encrypts and decrypts the low 4 bytes and high 4 bytes, respectively (Figure 9(c)). After the decryption and integrity check, RegVault II applies `or` operation to assemble the original 64-bit data (Line 11). In this way, RegVault II protects both the confidentiality and integrity of 64-bit data.

Developers can choose any protection options from our general scheme for their security-sensitive data.

5.5.2 *Control Data.* To defeat kernel control-flow attacks, RegVault II randomizes kernel control data in memory to prevent code pointers (return addresses and function pointers) from being controlled by attackers. Even when corrupting the in-memory representations, attackers cannot control the decrypted code pointers in registers. To further defeat the spatial substitution attacks, RegVault II encrypts code pointers with different tweaks.

Return address: RegVault II extends the compiler to instrument return address loading and storing automatically. More specifically, RegVault II encrypts all 8 bytes of a return address with the stack pointer as the tweak in the prologue. As a reverse procedure, the return address is loaded and decrypted in the epilogue before returning. Moreover, RegVault II adds a per-thread key field to the `thread_info`, which is initialized at thread forking. Therefore, each thread is encrypted with a unique key.

Function pointer: RegVault II detects the function pointer usages and inserts the encryption and decryption instructions automatically. RegVault II uses a different key to randomize kernel function pointers. Moreover, the storage address of a function pointer is used as the tweak to diversify the randomization. Let us use Figure 10 to illustrate the randomization. Line 3 in Figure 10(a) shows that a function pointer is loaded and then stored in another location. Figure 10(b) and (c) shows the corresponding instrumentation. The loaded value is decrypted using the loading address as the tweak (Lines 2 and 3). Then, it is encrypted using the new target address as the tweak before being stored in memory (Lines 4 and 5). To identify function pointers, RegVault II adopts an over-approximate approach by using the function pointer type and regarding all `void *` as function pointers. Moreover, to support address-based randomization, RegVault II leverages type information to identify function pointers copied by `memcpy/memmove/kmemdup` functions and re-randomize them based on the destination address.

5.5.3 *User Credentials.* The attackers often corrupt the uid/gid fields of `cred` struct to escalate their privileges [69]. RegVault II uses the annotation-based automatic instrumentation to randomize user credentials. The annotation and instrumentation are already covered in Sections 5.2 and 5.3. Note that `uid` is a 32-bit unsigned integer, RegVault II extends it to 64-bit and uses the top bit for integrity checking. Here, we randomize all data fields within `cred`, except the `usage` and RCU fields.

5.5.4 *Security Feature States.* Linux kernel security features often save their states in memory, which can be corrupted to bypass the protection. Using SELinux as an example, the SELinux states, `selinux_enforcing` and `ss_initialized`, are security-critical global variables, controlling the on/off of SELinux. Unfortunately, these variables are globally writable and can be located and

```c
static void process_one_work(struct worker *worker, struct work_struct *work) {
    ......
    worker->current_func = work->func;
    ......
}
```

(a) Function pointer load and store in C code.

```
1 ld    a0, 24(s0)          ; load encrypted work->func
2 addi  a1, s0, 24          ; address of work->func
3 crdak a0, a0, a1, 0, 7    ; decrypt work->func
4 addi  s3, s1, 24          ; address of worker->current_func
5 creak a0, a0, s3, 0, 7    ; encrypt worker->current_func
6 sd    a0, 24(s1)          ; store encrypted worker->current_func
```

(b) Function pointer load and store in RISC-V assembly code. Lines 2 and 3 show the decryption of function pointer `work->func`, and Lines 4 and 5 show the encryption of function pointer `worker->current_func`.

```
1 ldr   x1, [x4, #24]     ; load encrypted work->func
2 add   x2, x4, #24       ; address of work->func
3 crdak x1, x2, #0, #7    ; decrypt work->func
4 add   x6, x5, #24       ; address of worker->current_func
5 creak x1, x6, #0, #7    ; encrypt worker->current_func
6 str   x1, [x5, #24]     ; store encrypted worker->current_func
```

(c) Function pointer load and store in Arm assembly code. Lines 2 and 3 show the decryption of function pointer `work->func`, and Lines 4 and 5 show the encryption of function pointer `worker->current_func`.

Fig. 10. Function pointer randomization in RegVault II.

corrupted by the attacker easily [69]. Actually, researchers have demonstrated how to overwrite the `ss_initialized` to bypass SELinux protection [71].

In the recent Linux kernel, these variables are gathered in a global struct named `selinux_state`. Though the `ss_initialized` was renamed to `selinux_state.initialized`, the code logic remains the same, leaving the same weak spot. Therefore, RegVault II uses the annotation and the instrumentation to randomize all fields inside `struct selinux_state` (except the lock fields) to enhance the vulnerability-resilience of SELinux. The steps are the same with the `cred.uid` protection. Moreover, `selinux_state.initialized` is a bool, allowing RegVault II to use the top bits for the integrity checking.

5.5.5 Memory Management Data. Page tables are essential components in the kernel that manage memory access permissions. Unfortunately, page tables are globally writable in kernel mode, allowing attackers to exploit kernel vulnerabilities to manipulate page tables and disable memory protection [38, 59]. Theoretically, RegVault II could prevent such attacks by encrypting the entire page table, but this would require the MMU to support automatic decryption of page table entries during the page table walk. This would introduce the dual overhead of page table encryption and virtual address translation. To minimize the performance overhead of page table protection, RegVault II proposes to randomize every PGD pointer referencing the root of a page table hierarchy and hide page table locations.

RegVault II instruments all store-sites and load-sites of PGD pointers by annotating the `pgd_t` type and instructing our compiler to automatically insert encryption and decryption instructions. To defeat substitution, RegVault II uses the storage address of PGD pointers as the tweak to diversify the randomization. Moreover, to prevent attackers from locating page tables allocated statically and during the early boot phase, RegVault II re-allocates these page tables and page table entries and updates all references to them.

5.5.6 Cryptographic Keys and Intermediate Results. Most software-based encryption engines leave cryptographic keys unprotected in memory. Moreover, the intermediate cryptographic results may be leaked due to register spilling, allowing attackers to infer cryptographic keys. Therefore, to harden the existing software-based encryption algorithms, we use RegVault II cryptographic primitives to protect cryptographic keys and intermediate results.

RegVault II ensures that cryptographic keys are always encrypted in memory. To achieve this, during key setup phases, RegVault II encrypts these keys before storing them. After that, in encryption and decryption functions, RegVault II inserts decryption instructions immediately after all key loadings. Moreover, to randomize intermediate results, RegVault II annotates the encryption and decryption functions so that all spilled registers are encrypted and no intermediate results are leaked. As a proof of concept, we apply RegVault II to the AES engine in the Linux kernel `crypto` subsystem, and RegVault II successfully prevents disclosure related to AES keys, as detailed in Section 6.1.1.

5.6 Key Management

Master key initialization is done in machine mode by the bootloader. After initialization is done, the bootloader de-privileges to supervisor mode and starts to boot the kernel. As a result, the kernel can only execute `cremk` or `crdmk` but cannot read or write the master key register directly.

For the general keys, each thread might use a different key value, termed the *per-thread key*. RegVault II stores the per-thread key in `thread_info` of each thread by creating two 64-bit integers to store the high and low bits of the 128-bit key value. When the thread forks, RegVault II uses the master key to generate the per-thread key and encrypts it before storing it in the `thread_info`. During the context switch, the per-thread key is decrypted and loaded into the key registers. In this way, RegVault II ensures that the per-thread key in memory is always encrypted.

6 Evaluation

In this section, we evaluate RegVault II from both security and performance perspectives. We implement a prototype of RegVault II on Linux kernel v6.6.2 (with 750 lines of code changes), compiled by our extended LLVM/Clang 11 (with about 4,000 lines of code changes) with default configurations and 64-bit data integrity check. RegVault II focuses on OS kernel data protection. Therefore, the userspace parts of the testing environment, such as runtime libraries and benchmarks themselves, are not instrumented. For security analysis, we run our prototype and perform penetration tests under QEMU. For RISC-V performance analysis, we evaluate RegVault II on the Xilinx Virtex-7 FPGA VC707 board with our extended Rocket core (100 MHz) and 1 GB DDR3 memory. For Arm performance evaluation, we extend QEMU v8.1.1 to simulate the cryptographic instructions, with 8 GB of memory allocated.

6.1 Security Analysis

To thoroughly evaluate the security of RegVault II, we first perform penetration tests on RegVault II using real-world attacks. We then discuss two RegVault II specific attacks.

6.1.1 Penetration Tests. To test the effectiveness of RegVault II against memory corruption and memory disclosures, we perform penetration tests using the RIPE attack suite [78] and real-world attacks listed in Table 3. More specifically, we port RIPE to the RISC-V Linux kernel and simulate ❶ROP [64], ❷JOP [16], ❸data corruption attack, and ❹data disclosure attack. In addition to RIPE, we also develop additional real-world attacks against RegVault II, including ❺privilege escalation by corrupting `cred.uid` [71], ❻SELinux bypass by corrupting `selinux_state.initialized` [71], ❼interrupt context corruption by tampering with a register in the saved interrupt context, and ❽spatial code pointer substitution by replacing an encrypted function pointer with another in a different address.

Table 3. Penetration Test Results

Attacks	Defenses	
	Linux Kernel v6.6.2	RegVault II Protection
❶Return-Oriented Programming	X	√
❷Jump-Oriented Programming	X	√
❸Sensitive Data Corruption	X	√
❹Sensitive Data Leak	X	√
❺Privilege Escalation	X	√
❻SELinux bypass[71]	X	√
❼Interrupt Context Corruption	X	√
❽Spatial Code Pointer Substitution	X	√

The Linux kernel built under `defconfig` is vulnerable to the attacks listed, while RegVault II stops all of them successfully.

The test results in Table 3 show that RegVault II can defeat all the above attacks. More specifically, the address-based randomization scheme makes unauthorized writes unpredictable (defeating ❶,❷,❸,❺,❻), prevents data leaks (defeating ❹), and thwarts spatial substitution attacks(defeating ❽). Moreover, interrupt context corruption can be detected by our chain-based interrupt context protection (❼). In sum, RegVault II can stop state-of-the-art attacks against annotated sensitive data.

6.1.2 Time-of-derandomize-to-time-of-use Attack. In RegVault II, the data de-randomization and the use are in different instructions. This non-atomic design gives the attacker chances to launch the time-of-derandomize-to-time-of-use attack. To address this problem, we propose to randomize the interrupt context, as detailed in Section 5.4.1. As a result, even though the interrupt can happen between decryption and use, all registers saved to memory are encrypted, defeating any attempts to leak or corrupt the plaintext. Similarly, the gap between the data randomization and the data use is also protected by the interrupt context randomization.

6.1.3 Cold Boot Attack and Memory Sniff. Attackers can exploit physical attacks to extract sensitive memory data, employing techniques such as cold boot attacks or memory sniffing. RegVault II mitigates these threats by ensuring that confidential data is always stored in memory in an encrypted form, rendering it indecipherable to attackers without access to the encryption key. Moreover, encryption keys are securely retained exclusively within the processor's registers or memory in an encrypted form, making them inaccessible to external threats and effectively safeguarding the confidentiality and integrity of memory data.

6.1.4 Cryptographic Lookaside Buffer (CLB) Timing Side-channel. The attacker may leverage the CLB timing to launch side-channel attacks (short for CLB timing attack). Correspondingly, RegVault II introduces two defensive measures. First, in the current design, the user space does not have the privilege to run the cryptographic primitives. As a result, it cannot probe the CLB directly. Second, the hit-and-miss of the memory cache usually introduces hundreds of cycles of timing differences. In contrast, the timing difference for CLB is only several cycles, which is hard to measure accurately. Therefore, it is difficult to launch the CLB timing attack successfully.

6.1.5 Transient Execution Attack. Attackers can exploit transient execution vulnerabilities to launch transient execution attacks, including Meltdown and Spectre. RegVault II provides robust defenses against such attacks through multiple security measures. First, RegVault II mitigates traditional transient execution attacks, including Spectre and Meltdown, by encrypting secrets using data randomization. This ensures that even if an attacker successfully executes such an attack, they can only leak ciphertext, rendering the extracted data useless. Second, while attackers may attempt to bypass randomization protections within the transient window–such as decrypting secrets and leaking them through cache covert channels–these attacks are highly challenging in practice. RegVault II ensures that protected data does not involve code segments that decrypt secrets and immediately perform memory operations with the secret's plaintext, preventing attackers from leveraging existing memory code to initiate conventional cache-based side-channel leaks. For additional security, RegVault II can integrate software-level defenses, such as inserting fence instructions, to prevent transient window execution risks stemming from user-defined data or specific processor implementations. Third, RegVault II's CLB is designed to resist transient execution attacks. The updated results of CLB entries are temporarily cached in the Reorder Buffer (RoB) or other hardware buffers and are not applied immediately. These updates are only committed once encryption and decryption instructions are finalized, ensuring that sensitive data is never exposed in the CLB during transient execution windows. Furthermore, the CLB itself cannot be exploited as a covert channel for leaking secret information.

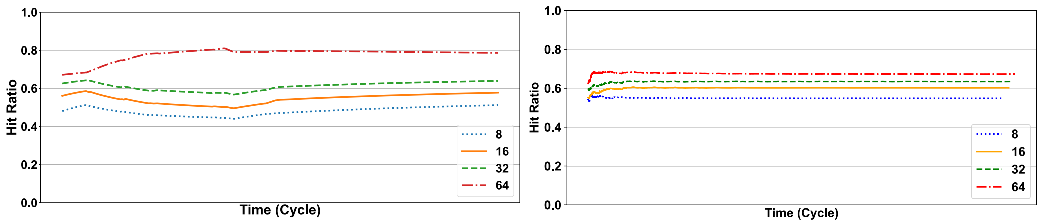
6.1.6 EPF-style Attack. RegVault II can effectively disrupt EPF-style attacks[36] by leveraging register-level encryption to secure critical eBPF data structures. EPF-style attacks exploit weaknesses in the BPF infrastructure[47] to inject or reuse malicious payloads. At first, BPF programs cannot read or write the crypto-key registers. Further, the eBPF data structures can be protected by RegVault II, thus preventing the attacker from manipulating critical pointers. Even if an attacker were to execute an EPF attack successfully, the cryptographic protection provided by RegVault II prevents unauthorized modification of sensitive registers. In other words, while EPF targets the manipulation of kernel memory via BPF, RegVault II's register-level encryption disrupts such attacks by ensuring that any tampering with critical data would fail the decryption check.

6.2 Performance Evaluation

For performance evaluation, we first test the CLB hit ratio with different numbers of entries. Next, we evaluate the performance gain brought about by the SCIE-based implementation. Moreover, we use two micro-benchmarks and one macro-benchmark to thoroughly test RegVault II implementation with 8 CLB entries.

Accurately measuring runtime overhead on QEMU for Arm is challenging. Since QEMU only emulates functionalities rather than faithfully replicating CPU timing behavior, even single-cycle instructions exhibit variable execution times in emulation. As a result, there is no reference standard for determining the exact overhead introduced by QEMU when instrumented instructions consume a specific number of cycles in an actual CPU pipeline. A more precise alternative, cycle-accurate GEM5, suffers from excessively long emulation times, making it impractical for large-scale evaluations. To strike a balance between accuracy and efficiency, we estimate execution overhead by analyzing how QEMU **Tiny Code Generator** (**TCG**) translates emulated single-cycle Arm instructions into equivalent TCG operations. Specifically, we determine the upper bound N of equivalent TCG operations for a given instruction. We then simulate M cycles consumed by each RegVault II instruction by performing N basic TCG operations M times, approximating the real-world execution overhead within the QEMU emulation framework.

6.2.1 CLB Hit Ratio. We evaluated the CLB's hit ratio using a micro-benchmark (UnixBench) and a real-world application (Docker) as macro-benchmarks. Figure 11(a) shows that a CLB with just eight entries achieves a hit ratio of approximately 50%, which means that most decryption instructions can retrieve their results directly from the CLB without needing to perform actual



(a) CLB hit ratio during UnixBench.

(b) CLB hit ratio during Docker.

Fig. 11. RegVault II CLB hit ratio with different number of entries. The hit ratio reaches 80% with 64 entries during UnixBench on the RISC-V FPGA platform, while it reaches 70% during Docker on the Arm QEMU platform.

Table 4. FPGA Resource Overhead of Crypto-engine with Different Number of CLB Entries on the Extended RISC-V Rocket Core

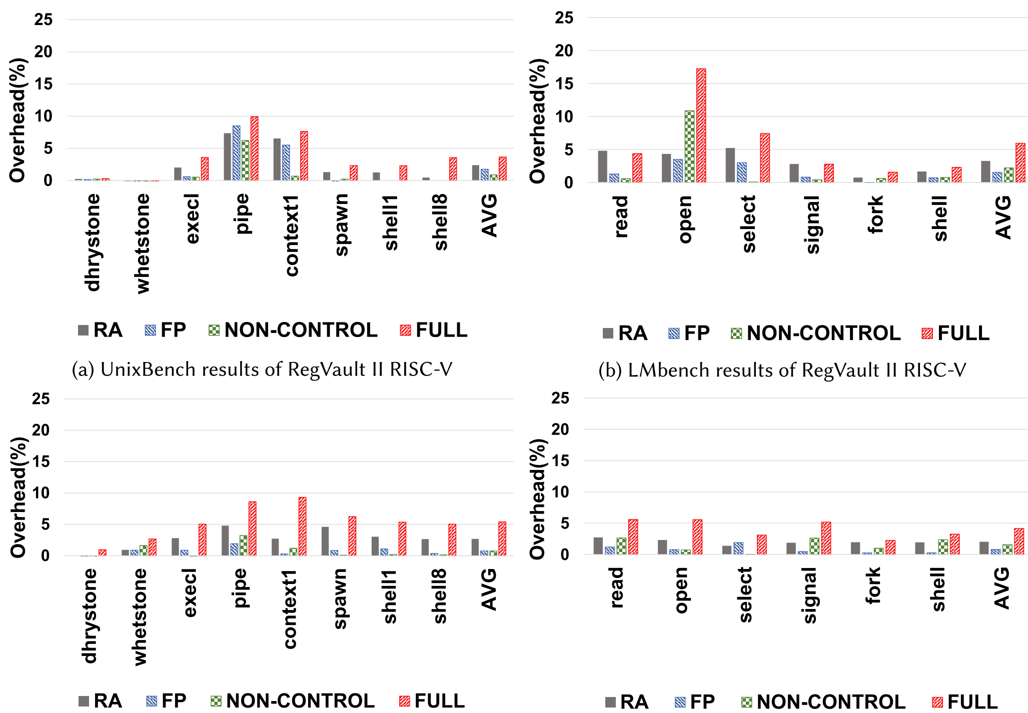
Entry Num	Slice LUT	LUT Ratio	Slice Regs	Reg Ratio	Energy Consum Ratio
8	1503	2.93%	1704	4.93%	0.33%
16	2876	5.47%	3274	9.07%	0.68%
32	4050	7.32%	6473	16.34%	1.28%
64	7375	12.21%	12685	21.86%	2.31%

decryption. As the number of CLB entries increases, the hit ratio improves, reaching 80% for 64 entries. Similarly, Figure 11(b) illustrates that in Docker, an 8-entry CLB achieves around a 55% hit ratio, which further increases to 70% for 64 entries. This demonstrates that the CLB maintains a high hit rate even in real applications.

While increasing the number of CLB entries can further enhance the hit ratio, a larger full-associative CLB also leads to higher hardware overhead and energy consumption. We measure the number and percentage of LUTs and **registers** (**Regs**) on FPGA slices and the energy consumed by the encryption engine using Vivado 2020.2. As shown in Table 4, the hardware overhead and energy consumption of the RISC-V FPGA implementation increase roughly linearly with the number of CLB entries. To balance performance and hardware efficiency, RegVault II prototype is implemented with a CLB containing eight entries, ensuring an optimal tradeoff between the hit ratio and the hardware cost.

6.2.2 Micro-benchmark Results. We select the UnixBench and the LMbench [49] for micro-evaluation. We test the performance overhead with four protection configurations: protecting **return addresses** (**RA**) only, protecting **function pointers** (**FP**) only, protecting non-control data, including all four types of **non-control** (**NON-CONTROL**) data, and full protection, including two types of control data and four types of non-control data (FULL) (Table 2). As shown in Figure 12, RegVault II RISC-V FPGA implementation introduces a 3.7% performance overhead for UnixBench and 6.0% for LMbench, while the Arm QEMU version incurs a 5.4% overhead for UnixBench and 4.2% for LMbench. Specifically, within the 1.6% performance impact of the NON-CONTROL protection mode, the (de)randomization of struct `selinux_state` contributes 0.19%, while `struct cred` accounts for 0.26%. Since these micro-benchmarks are primarily syscall-oriented, they provide an upper bound on performance overhead for userspace programs.

As expected, RegVault II incurs minimal runtime overhead in userspace arithmetic benchmarks such as Dhrystone and Whetstone (from UnixBench). In contrast, benchmarks that involve



(a) UnixBench results of RegVault II RISC-V

(b) LMbench results of RegVault II RISC-V

(c) UnixBench results of RegVault II Arm QEMU

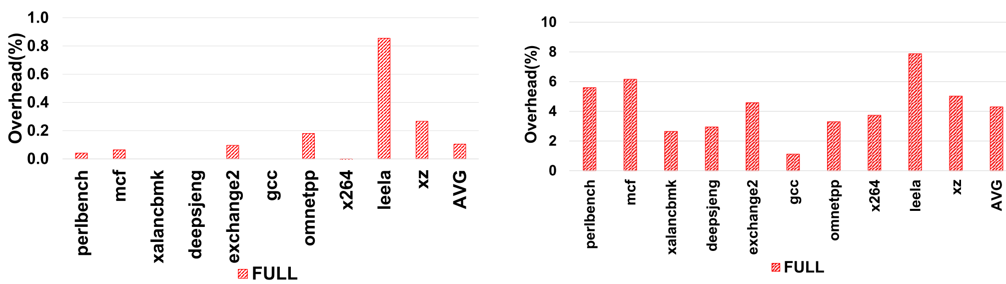
(d) LMbench results of RegVault II Arm QEMU

Fig. 12. RegVault II micro-benchmark results. The FULL protection mode of RegVault II RISC-V FPGA version has a 3.7% overhead for UnixBench and a 6.0% overhead for LMbench; the Arm QEMU version has a 5.4% overhead for UnixBench and a 4.2% overhead for LMbench.

frequent kernel interactions, such as syscall tests, exhibit relatively higher overhead. This is because cryptographic primitives in RegVault II are instrumented within the kernel; the runtime overhead scales with the length of the kernel function call chain, especially when RA protection is enabled. RA protection contributes the most overhead since nearly every kernel function is instrumented under this mode. However, in test cases involving frequent function pointer operations, such as `select`, the FP protection accounts for a significantly larger portion of the overhead.

Furthermore, the compilation of the Linux kernel is closely tied to optimization options, which are typically enabled to ensure successful compilation. When instrumenting the Linux kernel, adding more instrumentation instructions may prompt the compiler to apply more aggressive optimization strategies. For instance, after instrumenting 4 M of kernel code with FP protection, the compiler may perform deeper optimizations, resulting in some uninstrumented functions running faster than before. This effect may explain why certain test cases, such as spawn, exhibit an overhead slightly below zero. Additionally, while the FULL protection mode is composed of RA, FP, and NON-CONTROL protections, the more comprehensive optimizations applied in FULL protection can lead to cases where the observed performance overhead is lower than the sum of the individual RA, FP, and NON-CONTROL protection overheads.

6.2.3 Macro-benchmark and Application Results. We use the CINT test suite of SPEC CPU 2017 as the macro benchmark. As shown in Figure 13, the performance overhead is close to zero for RegVault II RISC-V and 4.3% for RegVault II Arm QEMU, indicating that RegVault II has a minimal



(a) SPEC CPU 2017 result of RegVault II RISC-V.

(b) SPEC CPU 2017 result of RegVault II Arm QEMU

Fig. 13. RegVault II macro-benchmark results. The RISC-V version achieves a close-to-zero overhead; The Arm QEMU version has a 4.3% overhead.

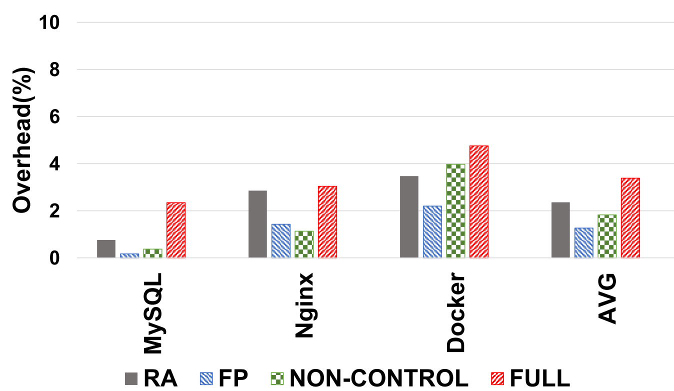


Fig. 14. RegVault II real-world application benchmark results. Arm QEMU version has a 3.4% overhead for FULL protection.

performance impact on common userspace programs. We further evaluate the performance of RegVault II on the Arm QEMU platform using MySQL [54], Nginx [58], and Docker [27]. As shown in Figure 14, the performance overhead is 3.4% for RegVault II Arm QEMU with FULL protection.

For MySQL (version 8.0.41), we measure read-write throughput as the evaluation metric, employing `sysbench` [3] to simulate remote client workloads. The experiment involves creating 10 database tables, each containing 10,000 records, and running 10 concurrent client threads. Using the OLTP read-write benchmark, we execute 20 independent runs for each Linux kernel protection mode. The results indicate that RegVault II's FULL protection mode on the Arm QEMU platform incurs a 2.3% reduction in read-write throughput.

For Nginx (v1.18.0), we use Apache Benchmark (`ab`) [29], generating 10,000 HTTPS requests, each retrieving a 1 KB file from the target server, with a concurrency level of 100 simultaneous clients. After an initial warm-up phase to stabilize the system, we conduct 20 independent experimental trials and compute the arithmetic mean of the collected data. The results show that FULL protection mode on the Arm QEMU platform incurs a 3.0% performance overhead.

For Docker (v27.5.1), we evaluate system call latency during the cold start procedure, measuring the average duration of system calls under RegVault II's protection. After an initial warm-up phase, we conduct 20 independent trials and compute the arithmetic mean of the recorded execution times. The results show that FULL protection mode introduces a 4.8% overhead on Docker's system call execution time.

Table 5. Comparison with Recent Data Memory Protection Approaches

Approaches		Integrity	Confidentiality	Cross Architecture	Fine Granularity	Multiple Data Type Support	Performance
Software-based	Randomized Layout[44]	X	X	√	√	X	√
	Runtime Detection[1]	√	X	√	√	X	X
Hardware Feature-based	Arm PA [40]	√	X	X	√	X	√
	Arm MTE [5]	√	X	X	√	X	X
	Intel MPK [62]	√	X	X	X	X	√
	Virtualization[80]	√	X	√	X	X	√
Hardware Customization-based	Morpheus[32]	√	√	√	√	X	√
	RegVault II	√	√	√	√	√	√

[1]It means the approach supports multiple data types with specific protection modes targeting each type.
[2]We only discuss hardware-assisted virtualization.

6.3 Comparison with Kernel Data Protection

Compared to recent OS kernel data protection approaches [1, 32, 40, 62, 74, 80], RegVault II adopts the strongest threat model, where the adversary has arbitrary read and write accesses to kernel data. In contrast to prior work that relies on spatial obfuscation or integrity-only mechanisms, RegVault II also delivers low-overhead, register-grained confidentiality and integrity protection. Table 5 summarizes how RegVault II outperforms these methods.

Software-based approaches, such as randomized structure layout [44] and **Data and Pointer Prioritization** (**DPP**) [1], focus on selective data protection without depending on specific hardware features. However, they suffer from significant limitations. These techniques typically exhibit low entropy, making them vulnerable to brute-force attacks and side-channel leaks (such as KASLR bypasses). Furthermore, frameworks like DPP rely on rule-based heuristics and tools like **Address Sanitizer** (**ASan**) [70] to detect security violations, which results in considerable runtime overhead. Moreover, such approaches mainly emphasize data integrity and often fail to safeguard the confidentiality of sensitive data.

Hardware feature-based data protection mechanisms leverage existing hardware features, such as Arm PA [40], Intel MPK [62], and virtualization-based memory safety enforcement[80] to protect data without modifying the underlying hardware. Although these solutions can boost performance, their reliance on specific architectural extensions restricts their applicability across diverse platforms. Moreover, Arm PA only targets the integrity of pointers; Intel MPK relies on page tables to provide coarse-grained protection. In contrast, RegVault II introduces unified hardware interfaces that enable cross-ISA adaptability while providing precise confidentiality and integrity protection for various types of sensitive kernel data.

Hardware customization-based approaches [32, 74] combine the flexibility of software with the efficiency of hardware, enabling robust and adaptable security mechanisms. RegVault II distinguishes itself from other hardware-software co-design approaches by offering precise, register-grained protection for both confidentiality and integrity of kernel data. This advanced capability allows RegVault II to protect small-grained runtime data, even when it is co-located with non-critical data, thereby mitigating sophisticated data attacks.

6.4 Limitation and Discussion

6.4.1 Temporal Attacks. Though RegVault II can defeat spatial substitution attacks effectively, it is still vulnerable to temporal substitution attacks. However, the proposed chain-based interrupt context protection can defeat the temporal substitution on the individual register. Moreover, techniques designed for combating temporal attacks [28] can be adopted by RegVault II.

6.4.2 Data-flow Tracing. Based on the annotation, RegVault II provides field-sensitive protection. However, RegVault II cannot trace sensitive data flow. That is, RegVault II does not protect in-memory objects without annotation, even if sensitive data flows to them. Our rationale is that most security-related data is located in certain structs (e.g., `cred`), and most corruption attacks target fields within these structs. Therefore, field-sensitive protection is capable of defeating these attacks. Furthermore, a comprehensive data tracing algorithm inevitably relies on the precise points-to analysis, which cannot be scalable to kernel [82].

6.4.3 Performance Concerns. Though RegVault II randomizes return addresses and function pointers, branch prediction accuracy is not impacted. The reason is that **branch prediction units** (**BPUs**) of modern processors obtain and store the target addresses before the encryption. For the subsequent prediction, the BPUs use the stored history for the prediction, which can correctly predict a return address and the function pointer ahead of the decryption.

6.4.4 Real-world Systems Integration. On Arm platforms with PA extension, RegVault II can leverage the same cryptographic engine used by PA instructions. While PA truncates the encrypted value and stores it in the unused higher bits, RegVault II writes the full ciphertext directly to the register. Consequently, RegVault II does not require an additional cryptographic engine, simplifying its integration with existing hardware.

Integrating RegVault II into real-world systems would require targeted modifications to existing operating system kernels, particularly in the areas of operations on critical kernel data. To mitigate potential impacts on maintainability, these modifications are designed as modular, well-abstracted components that interact seamlessly with the current kernel architecture. By encapsulating the main instrumentation procedure in an LLVM-based compiler, developers can limit the scope of changes and preserve the overall manageability of the system.

7 Related Work

Data space randomization (DSR): DSR randomizes the in-memory representation of objects to make unauthorized accesses unpredictable, thus stopping memory leaks and corruption attacks with a high probability. PointGuard [23] randomizes pointers using simple XOR with a global mask, which is not secure enough, as the leaked masked data allows attackers to recover the mask. Data space randomization works [15, 18] leverage pointer analysis to partition the data into different equivalence classes and assign a random mask to each class to XOR data. Even so, they are still susceptible to the disclosure of memory-resident XOR masks. To protect the secret XOR masks, HARD [14] achieves context-aware data partitioning and stores the masks in a protected memory key table. Moreover, CoDaRR [67] further proposes to re-randomize the masks periodically to protect the XOR masks. Besides, Potteiger et al. implemented a DSR for cyber-physical systems and achieved corruption detection by using a redundant backup [63]. Nevertheless, all of these works suffer from memory disclosures due to the XOR-based encryption. In contrast, RegVault II provides lightweight yet cryptographically strong randomization primitives by implementing a hardware cryptographic engine based on the QARMA cipher. Besides, using the tweak as an extra input, RegVault II is able to achieve fine-grained data randomization (e.g., based on address).

Selective data protection: Palit et al. propose two selective data protection defenses to protect the selected sensitive data [60, 61]. Similar to data randomization, these implementations use cryptographic methods to encrypt sensitive data, such as cryptographic keys and user passwords, to prevent information leakage. [60] mainly uses static analysis to track sensitive data flow and inserts AES operations to prevent arbitrary access to the protected data. DynPTA [61] also uses AES for the encryption. It combines static analysis with dynamic data flow tracing for better accuracy and performance. Ginseng [81] uses static taint analysis to identify all sensitive variables. It encrypts and decrypts this data with AES operations using a higher privilege mode's system call. **Code-Pointer**

Integrity (CPI) [37] allocates an isolated safe memory region for code pointers and all data pointers used to access code pointers. CCFI [46] generates **message authentication codes** (**MACs**) based on AES-NI instructions and uses MACs to validate the integrity of code pointers.

As mentioned before, AES is not suitable for runtime encryption on register-sized data. The AES-NI-based CCFI[46] introduces 3-18% performance overhead on the application and an average of 52% overhead for all benchmarks in SPEC CPU 2006. RegVault II provides atomic and lightweight cryptographic primitives, which are designed for register-grained data. As a result, the performance overhead of RegVault II is as low as up to 6.0% for micro-benchmarks and is close to zero for the macro-benchmark in the RISC-V version, while 5.4% for micro-benchmarks, 4.3% for the macro-benchmark, and 3.4% for the application-benchmark in the Arm version.

Hardware-based data isolation: There are also hardware-based data isolation studies. PixelVault [76] leverages the GPU to enhance the security of cryptographic operations, keeps cryptographic keys, and carries out cryptographic operations exclusively on the GPU registers and instruction cache. By conducting cryptographic computations entirely within the GPU and storing cryptographic keys exclusively in GPU registers, PixelVault isolates sensitive operations from the main system memory and CPU, thereby reducing the attack surface available to potential adversaries. While considering security features provided by the CPU, HDFI [73] is an efficient hardware-based data isolation mechanism that isolates different data flows with low overhead by using tagged memory. Morpheus [33] also uses tagged memory to tag different domains and perform high-frequency churn on them, making the system impractically difficult to penetrate. ERIM [75] uses the Intel MPK extension to provide efficient in-process data isolation for user space programs. EPK [35] and VDom [79] virtualize Intel MPK to offer scalable in-process isolation. Also leveraging Intel MPK novelly, Safeslab [53] provides a heap-hardening approach to efficiently block the dangling pointers and thus mitigate the use-after-free vulnerabilities with a drastic overhead decrease. ISLAB [52] utilizes Intel SMAP to protect the integrity of memory management metadata and in-kernel sensitive data structures. In particular, ISLAB notices the interrupt state and keeps the key registers in an isolated stack. Our approach is based on a chained en(de)cryption, which does not need a special secure stack and instruction rewriting while introducing a low cryptographic operation overhead. In addition, hypervisors [13, 43, 65, 72, 80] and trusted execution environments [9, 10, 30, 31, 51] are also used for in-process or kernel critical data protection. These works are orthogonal to our work.

Hardware-based cryptographic primitives: Intel, Arm, and many other vendors provide hardware extensions for common steps in cryptographic algorithms, such as AES and SHA [6]. One complete encryption or decryption usually needs several instructions in the extension, which breaks the atomicity of the whole encryption.

ARMv8.3 introduced PA [66], a hardware-assisted mechanism to sign and authenticate pointers. Leveraging PA, PARTS [41] protects code and data pointers for user space programs, Camouflage [26] protects a fraction of kernel function pointers, and PACStack [40] achieves a context-sensitive return address protection. But Arm PA is designed to enforce pointer integrity and thus cannot provide confidentiality protection for general data. However, Arm PA needs to store the authentication code in the unused bits of the pointers, and thus cannot be used for general data randomization.

Cryptoraptor [68] is a domain-specific high-performance processor covering a wide range of cryptographic algorithms. However, it is not integrated into the CPU core. Recryptor [83] is a configurable cryptographic processor that aims at in-memory computing. Techniques like Intel **Total Memory Encryption** (**TME**) use a crypto-engine to encrypt memory. These techniques target specific scenarios with different threat models and thus cannot be used for general data randomization.

Memory safety-based protection schemes aim at eliminating memory bugs in the first place. To achieve spatial safety, SoftBound [55], CCured [57], and Baggy Bounds Checking [4] enforce bound checking for every memory access. CETS [56] further provides temporal safety by allocating a unique identifier with each object and associating it with related pointers. Nevertheless, all the above schemes require extra checks on every memory access, leading to prohibitive performance overhead.

Therefore, to reduce the performance overhead, researchers apply memory safety to sensitive data only. DataShield [19] divides process memory into sensitive and non-sensitive parts and enforces precise spatial and temporal safety checks only on sensitive data. Similarly, ConfLLVM [17] partitions the memory into private/public regions and inserts runtime checks to ensure every pointer points to the correct memory region inferred by the compiler. But still, both of them incur relatively high overheads (more than 10% on SPEC).

8 Conclusion

This article presents RegVault II, a hardware-assisted selective data randomization scheme for OS kernels. To achieve RegVault II, at the hardware level, we design and implement the novel hardware primitives, supporting cryptographically strong encryption for register-grained data. To further reduce the performance overhead, we integrate the cryptographic operations into the pipeline. We also introduce a new type of cache to buffer the cryptographic results to avoid redundant cryptographic operations.

At the software level, RegVault II develops the annotation-based approach to allow kernel developers to mark sensitive data. Based on the annotation, RegVault II extends the LLVM compiler to automatically instrument sensitive data stores and loads with encryption and decryption primitives, achieving randomization. In addition, to guarantee comprehensive randomization, RegVault II also develops the chain-based interrupt context protection to protect the interrupt context and the cross-call spill protection to protect the register spilling.

To demonstrate the effectiveness of RegVault II, we build a prototype of RegVault II to protect Linux kernel runtime data, including two types of control data and four types of non-control data. We also conduct a thorough evaluation of both the security and the performance of the prototype. The security analysis shows that RegVault II can defend against kernel data attacks effectively. The performance overhead of RegVault II is minimal, which is at most 6.0% for micro-benchmarks, and is close to zero for the macro-benchmark in the RISC-V version, while 5.4% for micro-benchmarks, 4.3% for the macro-benchmark, and 3.4% for the application-benchmark in the Arm version.

Acknowledgments

The authors would like to thank reviewers for their insightful comments. Those comments helped to reshape this article.

References

[1] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, and Danfeng (Daphne) Yao. 2023. Not all data are created equal: Data and pointer prioritization for scalable protection against data-oriented attacks. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, Anaheim, CA, 1433–1450. Retrieved from https://www.usenix.org/conference/usenixsecurity23/presentation/ahmed-salman

[2] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. *Breakthrough AES Performance with Intel AES New Instructions*. Technical Report. Intel. 11 pages.

[3] akopytov. 2025. Sysbench. Retrieved 19 May, 2025 from https://github.com/akopytov/sysbench

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada). USENIX Association, USA, 51–66.

[5] ARM. 2019. ARM Memory Tagging Extension. Retrieved 19 May, 2025 from https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf
[6] ARM. 2021. ARM Cryptography Extension. Retrieved 19 May, 2025 from https://developer.arm.com/documentation/ddi0514/g/introduction/about-the-cortex-a57-processor-cryptography-engine
[7] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html
[8] Roberto Maria Avanzi. 2017. The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency S-Boxes. *IACR Transactions on Symmetric Cryptology* 2017, 1 (2017), 4–44. DOI: https://doi.org/10.13154/tosc.v2017.i1.4-44
[9] 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* (Scottsdale, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 90–102. DOI: https://doi.org/10.1145/2660267.2660350
[10] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of the 2016 Network and Distributed System Security Symposium*. NDSS, San Diego, California, USA.
[11] Brandon Azad. 2020. iOS Kernel PAC, One Year Later. Retrieved 19 May, 2025 from https://i.blackhat.com/USA-20/Wednesday/us-20-Azad-iOS-Kernel-PAC-One-Year-Later.pdf
[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California). Association for Computing Machinery, New York, NY, USA, 1216–1225. DOI: https://doi.org/10.1145/2228360.2228584
[13] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA). USENIX Association, USA, 335–348.
[14] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M. Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. 2018. Hardware assisted randomization of data. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, Association for Computing Machinery, New York, NY, USA, 337–358.
[15] Sandeep Bhatkar and R. Sekar. 2008. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Paris, France). Springer-Verlag, Berlin, 1–22. DOI: https://doi.org/10.1007/978-3-540-70542-0_1
[16] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China). Association for Computing Machinery, New York, NY, USA, 30–40. DOI: https://doi.org/10.1145/1966913.1966919
[17] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. 2019. ConfLLVM: A compiler for enforcing data confidentiality in low-level code. In *Proceedings of the 14th EuroSys Conference 2019* (Dresden, Germany). Association for Computing Machinery, New York, NY, USA, Article 4, 15 pages. DOI: https://doi.org/10.1145/3302424.3303952
[18] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. 2008. *Data Randomization*. Technical Report. Technical Report TR-2008-120, Microsoft Research, 2008. Cited on.
[19] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates). Association for Computing Machinery, New York, NY, USA, 193–204. DOI: https://doi.org/10.1145/3052973.3052983
[20] RISC-V Community. 2021. RISC-V Cryptography Extension. Retrieved 19 May, 2025 from https://github.com/riscv/riscv-crypto
[21] Microsoft Corporation. 2019. A Proactive Approach to More Secure Code. Retrieved 19 May, 2025 from https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
[22] Nicolas T. Courtois and Josef Pieprzyk. 2002. Cryptanalysis of block ciphers with overdefined systems of equations. In *Proceedings of the Advances in Cryptology*. Springer, Berlin, 267–287.
[23] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointguardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC). USENIX Association, USA, 7.

[24] Asmit De, Aditya Basu, Swaroop Ghosh, and Trent Jaeger. 2019. FIXER: Flow integrity extensions for embedded RISC-V. In *Proceedings of the 2019 Design, Automation Test in Europe Conference Exhibition*. DATE, Florence, Italy, 348–353. DOI: https://doi.org/10.23919/DATE.2019.8714980

[25] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. 2020. PHMon: A programmable hardware monitor and its security use cases. In *Proceedings of the 29th USENIX Conference on Security Symposium.* USENIX Association, USA, Article 46, 18 pages.

[26] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. 2020. Camouflage: Hardware-assisted CFI for the ARM Linux kernel. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference* (Virtual Event, USA). IEEE Press, USA, Article 224, 6 pages.

[27] Docker. 2025. Docker. Retrieved 19 May, 2025 from https://www.docker.com/

[28] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal memory safety via robust points-to authentication. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, USA, 1037–1054. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade

[29] The Apache Software Foundation. 2025. ab. Retrieved 19 May, 2025 from https://httpd.apache.org/docs/2.4/programs/ab.html

[30] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening just-in-time compilers with SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA). Association for Computing Machinery, New York, NY, USA, 2405–2419. DOI: https://doi.org/10.1145/3133956.3134037

[31] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-process memory isolation extension. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) . USENIX Association, USA, 83–97.

[32] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 469–484. DOI: https://doi.org/10.1145/3297858.3304037

[33] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, et al. 2019. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 469–484. DOI: https://doi.org/10.1145/3297858.3304037

[34] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference* (San Francisco, CA, USA). IEEE Press, USA, 769–774. DOI: https://doi.org/10.1109/DAC18074.2021.9586216

[35] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and efficient memory protection keys. In *Proceedings of the 2022 USENIX Annual Technical Conference* (Carlsbad, CA, USA). USENIX Association, USA, 609–624.

[36] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. EPF: Evil packet filter. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX Association, Boston, MA, 735–751. https://www.usenix.org/conference/atc23/presentation/jin

[37] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Broomfield, CO, 147–163. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[38] MWR Labs. 2014. Windows 8 Kernel Memory Protections Bypass. Retrieved 19 May, 2025 from https://labs.withsecure.com/publications/windows-8-kernel-memory-protections-bypass

[39] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. 2011. A cryptanalysis of PRINTcipher: The invariant subspace attack. In *Proceedings of the Advances in Cryptology*. Phillip Rogaway (Ed.), Springer, Berlin, 206–221.

[40] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. 2021. PACStack: An authenticated call stack. In *Proceedings of the USENIX Security*. USENIX Association, USA. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand

[41] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, 177–194. Retrieved from https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[42] Moses Liskov, Ronald L. Rivest, and David Wagner. 2002. Tweakable block ciphers. In *Proceedings of the Advances in Cryptology*. Moti Yung (Ed.), Springer, Berlin, 31–46.

[43] 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* (Denver, Colorado, USA). Association for Computing Machinery, New York, NY, USA, 1607–1619. DOI: https://doi.org/10.1145/2810103.2813690

[44] LWN. 2017. Randomizing Structure Layout. Retrieved 19 May, 2025 from https://lwn.net/Articles/722293/

[45] Ben Marshall, G. Richard Newell, Dan Page, Markku-Juhani O. Saarinen, and Claire Wolf. 2020. The design of scalar AES instruction set extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 1 (2020), 109–136. DOI: https://doi.org/10.46586/tches.v2021.i1.109-136

[46] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA). Association for Computing Machinery, New York, NY, USA, 941–951. DOI: https://doi.org/10.1145/2810103.2813676

[47] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (San Diego, California). USENIX Association, USA, 2.

[48] Kerry McKay, Lawrence Bassham, Meltem Sönmez Turan, and Nicky Mouha. 2016. *Report on Lightweight Cryptography*. Technical Report. National Institute of Standards and Technology.

[49] Larry McVoy and Carl Staelin. 1996. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (San Diego, CA). USENIX Association, USA, 23.

[50] Larkin Mike. 2015. *Kernel $W\hat{X}$ Improvements In OpenBSD*. Technical Report. OpenBSD.

[51] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. MicroStache: A lightweight execution context for in-process safe region isolation. In *Proceedings of the Research in Attacks, Intrusions, and Defenses*. Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.), Springer International Publishing, Cham, 359–379.

[52] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. ISLAB: Immutable memory management metadata for commodity operating system kernels. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (Singapore, Singapore). Association for Computing Machinery, New York, NY, USA, 1159–1172. DOI: https://doi.org/10.1145/3634737.3644994

[53] Marius Momeu, Simon Schnückel, Kai Angnis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. Safeslab: Mitigating use-after-free vulnerabilities via memory protection keys. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA). Association for Computing Machinery, New York, NY, USA, 1345–1359. DOI: https://doi.org/10.1145/3658644.3670279

[54] MySQL. 2025. MySQL. Retrieved 19 May, 2025 from https://www.mysql.com/

[55] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for c. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* 44, 6 (2009), 245–258. DOI: https://doi.org/10.1145/1543135.1542504

[56] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler enforced temporal safety for C. *Proceedings of the 2010 International Symposium on Memory Management* 45, 8 (2010), 31–40. DOI: https://doi.org/10.1145/1837855.1806657

[57] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* 27, 3 (2005), 477–526. DOI: https://doi.org/10.1145/1065887.1065892

[58] Nginx. 2025. Nginx. Retrieved 19 May, 2025 from https://www.nginx.com/

[59] Enrique E. Nissim Nicolas A. Economou. 2016. Getting Physical: Extreme Abuse of Intel Based Paging Systems. Retrieved 19 May, 2025 from https://www.coresecurity.com/core-labs/articles/getting-physical-extreme-abuse-of-intel-based-paging-systems

[60] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. 2019. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA). Association for Computing Machinery, New York, NY, USA, 598–611. DOI: https://doi.org/10.1145/3359789.3359815

[61] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining static and dynamic analysis for practical selective data protection. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA.

[62] Dinglan Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. μSwitch: Fast kernel context isolation with implicit context switches. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 2956–2973. DOI: https://doi.org/10.1109/SP46215.2023.10179284

[63] Bradley Potteiger, Zhenkai Zhang, and Xenofon Koutsoukos. 2019. Integrated data space randomization and control reconfiguration for securing cyber-physical systems. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*. Association for Computing Machinery, New York, NY, USA, 1–10.

[64] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security and Privacy* 10, 6 (2012), 84–87.

[65] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective memory protection for kernel and user space. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. IEEE, 563–577. DOI: https://doi.org/10.1109/SP40000.2020.00041

[66] Inc. Qualcomm Technologies. 2017. Pointer Authentication on ARMv8.3. Retrieved 19 May, 2025 from https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[67] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous data space randomization against data-only attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan). Association for Computing Machinery, New York, NY, USA, 494–505. DOI: https://doi.org/10.1145/3320269.3384757

[68] Gokhan Sayilar and Derek Chiou. 2014. Cryptoraptor: High throughput reconfigurable cryptographic processor. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California). IEEE Press, San Jose, California, 154–161.

[69] INetCop Security. 2016. New Reliable Android Kernel Root Exploitation Techniques. Retrieved 19 May, 2025 from http://powerofcommunity.net/poc2016/x82.pdf

[70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) . USENIX Association, USA, 28.

[71] Di Shen. 2017. Defeating Samsung KNOX with Zero Privilege. Retrieved 19 May, 2025 from https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege.pdf

[72] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen. In *Proceedings of the NDSS*. NDSS, San Diego, CA, USA.

[73] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, IEEE, San Jose, CA, USA.

[74] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. 2023. Multi-Tag: A hardware-software co-design for memory safety based on multi-granular memory tagging. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia). Association for Computing Machinery, New York, NY, USA, 177–189. DOI: https://doi.org/10.1145/3579856.3590331

[75] 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 *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, 1221–1238. Retrieved from https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[76] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA). Association for Computing Machinery, New York, NY, USA, 1131–1142. DOI: https://doi.org/10.1145/2660267.2660316

[77] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[78] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA). Association for Computing Machinery, New York, NY, USA, 41–50. DOI: https://doi.org/10.1145/2076732.2076739

[79] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. 2023. VDom: Fast and unlimited virtual domains on multiple architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 905–919. DOI: https://doi.org/10.1145/3575693.3575735

[80] Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. 2024. LightZone: Lightweight hardware-assisted in-process isolation for ARM64. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong). Association for Computing Machinery, New York, NY, USA, 467–480. DOI: https://doi.org/10.1145/3652892.3700786

[81] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping secrets in registers when you distrust the operating system.. In *Proceedings of the NDSS*. San Diego, CA, USA.

[82] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. 2019. PeX: A permission check analysis framework for Linux kernel. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA). USENIX Association, USA, 1205–1220.

[83] Yiqun Zhang, Li Xu, Qing Dong, Jingcheng Wang, David Blaauw, and Dennis Sylvester. 2018. Recryptor: A reconfigurable cryptographic Cortex-M0 processor with in-memory and near-memory computing for IoT security. *IEEE Journal of Solid-State Circuits* 53, 4 (2018), 995–1005. DOI: https://doi.org/10.1109/JSSC.2017.2776302

Received 2 July 2024; revised 23 March 2025; accepted 28 April 2025