RegVault: Hardware Assisted Selective Data Randomization for Operating System Kernels

Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu and Kui Ren
Zhejiang University, Hangzhou, China

ABSTRACT

This paper presents RegVault, a hardware-assisted lightweight data randomization scheme for OS kernels. RegVault introduces novel cryptographically strong hardware primitives to protect both the confidentiality and integrity of register-grained data. RegVault leverages annotations to mark sensitive data and instruments their loads and stores automatically. Moreover, RegVault also introduces new techniques to protect the interrupt context and safeguard the sensitive data spilling. We implement a prototype of RegVault by extending RISC-V architecture to protect six types of sensitive data in Linux kernel. Our evaluations show that RegVault can defend against the kernel data attacks effectively with a minimal performance overhead.

ACM Reference Format:
Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu and Kui Ren. 2022. RegVault: Hardware Assisted Selective Data Randomization for Operating System Kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, San Francisco, CA, USA, 6 pages. https://doi.org/10.1145/3489517.3530549

1 INTRODUCTION

Memory safety issues (i.e., memory corruption and disclosure) remain to be one of the biggest threats to modern systems, causing about 70% of all patches [8]. Exploiting the memory corruption, attackers can overwrite the control data to achieve control flow hijacking or the non-control data to achieve data-oriented attacks. Moreover, memory disclosure leaks return addresses or function pointers, allowing attackers to bypass KASLR easily. Unfortunately, these security-critical data are usually small-grained (i.e., register-grained) and mixed with non-security-critical data. Therefore, the page-grained (i.e., 4KB) protection is too coarse-grained and cannot be applied to protect these data.

To protect these small-grained data, researchers proposed data space randomization [6] to randomize the data in-memory representations with secret keys. Though the attacker can still overwrite the data, the corrupted data will be de-randomized to garbage values without correct secret keys. Therefore, data randomization is effective for defending against memory corruption and disclosure. However, due to the lack of hardware randomization primitives, existing data randomization schemes [6, 19] use XOR for the randomization and are vulnerable to memory disclosures. Moreover, research works also use AES primitives for control and non-control data encryption [7, 14, 16]. However, those AES primitives are designed for bulky data encryption and are not suitable for the register-grained data randomization. Therefore, *how to design cryptographically strong hardware randomization primitives for register-grained data and how to leverage these hardware primitives to achieve secure data randomization are still open problems.*

In this paper, we present an initial step in resolving these problems. We develop a hardware-assisted register-grained data randomization scheme named *RegVault* for OS kernels. To achieve RegVault, at the hardware level, we design and implement novel hardware primitives, supporting both the confidentiality and integrity protection for register-grained data. At the software level, we develop the annotation-based approach to mark the sensitive data and extend the LLVM compiler to automatically instrument the sensitive data loads and stores with hardware primitives. To guarantee comprehensive randomization, we also develop two new techniques—the *chain-based interrupt context protection* to protect the interrupt context and the *register spilling protection* to protect the register spilling.

To demonstrate the effectiveness of RegVault, we build a prototype of RegVault to protect the runtime data of the Linux kernel, including two types of control data and four types of non-control data. The security analysis indicates that RegVault can defend against the kernel data attacks effectively. The performance overhead introduced by RegVault is around 2.5% for micro-benchmarks and is close-to-zero for the macro-benchmark. In summary, this paper makes the following contributions.

- **New Hardware Primitives.** We design and implement novel hardware primitives, providing cryptographically strong *confidentiality and integrity* protection supports for register-grained data. We further propose the cryptographic lookaside buffer to reduce the performance overhead.
- **New Protection Techniques.** We propose *chain-based interrupt context protection* to protect interrupt contexts. We also propose *register spilling protection* to safeguard sensitive data spilling.
- **Prototype and Evaluation.** We implement a prototype of RegVault on Linux kernel v5.8.18, protecting six types of sensitive data. Our evaluations show that RegVault can defend against kernel data attacks effectively with a minimal performance overhead.

2 REGVAULT ARCHITECTURE

2.1 Threat Model and Assumptions

In our threat model, the attacker has full control of the user space, such as running any programs and invoking any system calls. The

Wenbo Shen is the corresponding author (shenwenbo@zju.edu.cn).
This work is partially supported by the National Natural Science Foundation of China (Grants No. 62002317 and No. U21A20464), by the National Key R&D Program of China (Grant No. 2020AAA0107700).

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530549

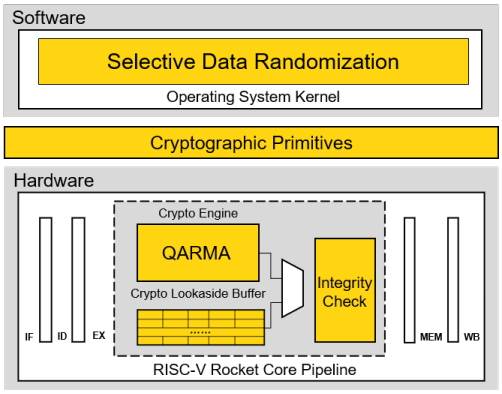


Figure 1: Overview of RegVault.

Table 1: RegVault cryptographic primitives.

Name	Mnemonic
context-aware register encrypt	`cre[x]k rd, rs[e:s], rt`
context-aware register decrypt	`crd[x]k rd, rs, rt, [e:s]`

attacker can also exploit kernel vulnerabilities to achieve arbitrary kernel memory read capability. Moreover, the attacker can overwrite or substitute any randomized data in kernel memory. However, the attacker cannot read or write the registers directly by using laser fault injections. We assume the kernel code is properly protected so that the attacker cannot corrupt existing code or inject new kernel code. Moreover, side-channel attacks are out of scope of this paper.

2.2 RegVault Overview

RegVault is a hardware-assisted selective data randomization architecture for operating system kernels. The basic idea of RegVault is to encrypt the annotated data in the memory and decrypt these data in registers. As shown in Figure 1, RegVault consists of the hardware and the software supports. For the hardware, RegVault extends 64-bit RISC-V core to implement lightweight cryptographic primitives, which are used to encrypt and decrypt the selected data to achieve selective data confidentiality and integrity protection (§2.3). For the software, RegVault leverages annotations and extends the compiler to instrument hardware primitives automatically (§2.4). To demonstrate the effectiveness, we apply RegVault to protect sensitive data in Linux kernel, which is discussed in §3.

2.3 Hardware Support

2.3.1 RegVault Cryptographic Primitives. Integrity enforcement based on traditional block ciphers is vulnerable to substitution attacks, i.e., swapping two encrypted values to bypass integrity check. Therefore, to provide strong integrity guarantee, RegVault chooses to use the tweakable block cipher QARMA [1] as the underlying cryptographic algorithm. QARMA takes 3 inputs: a 128-bit key, a 64-bit tweak and a 64-bit plaintext to produce a single 64-bit ciphertext. By changing the tweak, tweakable block ciphers can produce different ciphertexts even with the same key and plaintext. With the tweak as an extra input, RegVault is able to defeat substitution attacks against sensitive data by combining additional information (such as the storage address of the data) into the randomization.

RegVault extends the RISC-V instruction set to provide lightweight and fine-grained cryptographic primitives, as listed in Table 1. The extended instructions are termed as context-aware cryptographic instructions, as the tweak can be contextual information.

- `cre[x]k rd, rs[e:s], rt` is the encryption instruction. It selects bytes within range `[e:s]` (i.e., `s`-th to `e`-th bytes) from source register `rs` , encrypts them with the tweak in register `rt` and the key in key register `x`, and puts the ciphertext in destination register `rd`. Note that the bytes other than `[e:s]` are set to zero before encryption for integrity checking purpose.
- `crd[x]k rd, rs, rt, [e:s]` is the decryption instruction. It decrypts the value in `rs` with given tweak in `rt` and key `x`, and puts the plaintext in `rd`. Moreover, it checks whether the bytes other than `[e:s]` in plaintext remain zero. If not, the integrity check fails and an exception is raised.

Dedicated for kernel data randomization, these instructions are not executable in the user mode.

RegVault provides dedicated key registers (a master key `m` and 7 general keys `a-g`) by extending the CSRs. User space programs have no access to any key registers. Operating system kernels can write general key registers, but are not allowed to read them. Furthermore, even the kernels are prohibited to read or write the master key. Therefore, the master key can be used to encrypt other general keys saved in the memory.

Leveraging the range selection field `[e:s]`, these primitives provide flexible confidentiality and integrity protection. We choose three typical scenarios to demonstrate how to use these primitives to safeguard sensitive data, as shown in Figure 2.

- **Pointer**. RegVault encrypts and decrypts all bytes (i.e., with range `[7:0]`) in the pointers directly (Figure 2a). Therefore, any corrupted pointers in memory are decrypted into garbage values, pointing to illegal addresses. RegVault also uses this way to protect 64-bit data that do not require integrity protection.
- **32-bit data**. RegVault 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 2b Line 2). In the decryption, the upper 32-bit is used for the integrity check (Line 6). Therefore, RegVault protects both confidentiality and integrity for 32-bit data.
- **64-bit data**. RegVault splits 64-bit data as two 32-bit data, encrypts and decrypts the low 4 bytes and high 4 bytes respectively (Figure 2c). After the decryption and integrity check, RegVault applies `or` operation to assemble the original 64-bit data (Line 11). In this way, RegVault protects both the confidentiality and integrity for 64-bit data.

2.3.2 RegVault Crypto-Engine. To support the cryptographic primitives, RegVault implements a crypto-engine in the RISC-V pipeline. Before executing a cryptographic instruction, the engine first checks its executability according to current privilege level. For the encryption instruction, the engine first constructs the plaintext according to the source register and the selected range, and then performs the cryptographic operation. For the decryption instruction, the engine first decrypts the value in the source register, and then checks whether the bytes outside the selected range are zero. A failed integrity check raises an exception.

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

(a) Pointer randomization.

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

(b) 32-bit data randomization with integrity.

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

(c) 64-bit data randomization with integrity.

Figure 2: Data randomization using RegVault primitives. Highlight lines are instrumented primitives.

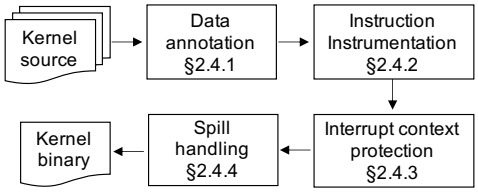


Figure 3: RegVault kernel data randomization process.

2.3.3 Cryptographic Lookaside Buffer. To substantially reduce the overhead of frequently-invoked cryptographic operations, we propose to integrate a cache, named *cryptographic lookaside buffer* (CLB), into the crypto-engine to hold recently-calculated results. When executing cryptographic instructions, RegVault crypto-engine first looks up CLB, before performing the multi-cycle operations.

CLB structure: CLB is a fully-associative cache, consisting of a configurable number of entries. Each entry of the CLB contains 6 fields: the metadata for replacement, a valid bit `v` marking whether the entry is valid, a key selection index `ksel`, the tweak, the plaintext, and the ciphertext.

CLB operations: To reduce the entry size, CLB stores 3-bit key selection indices instead of 128-bit keys. Therefore, when a key register is updated, the stale CLB entries with the corresponding `ksel` are invalidated. When querying, the CLB checks all entries and sends the result to the pipeline directly if a valid matching entry is found. Otherwise, the crypto-engine performs the cryptographic operation and updates the CLB with the newly generated result. If all entries are valid, CLB evicts one based on the least-recently-used (LRU) replacement policy.

2.4 Software Support

Figure 3 illustrates the overall flow of RegVault kernel data randomization. With the kernel source code as the input, RegVault first annotates the data types that need to be randomized (§2.4.1). After that, RegVault instruments the randomized data loads and stores with cryptographic primitives (§2.4.2). Finally, to defeat data leak and corruption on spilling, RegVault protects both the interrupt context (§2.4.3) and register spilling (§2.4.4). The following sections describe these steps in detail.

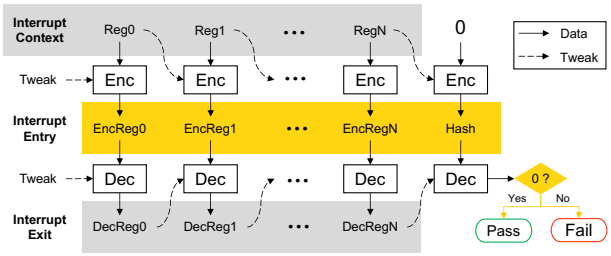


Figure 4: RegVault interrupt context protection.

2.4.1 Data Annotation. Before instrumentation, RegVault needs to know which data needs to be randomized. Inspired by the annotation `__randomize_layout` used in Linux kernel currently, RegVault provides two annotation macros—`__rand` for confidentiality only and `__rand_integrity` for both confidentiality and integrity protection. For example, to protect `uid` in `cred`, the developer annotates it as `kuid_t uid __rand_integrity`. These macros set storage sizes and alignments properly, and add compiler attributes that are recognized by RegVault during the compiling stage. Note that these macros are field-sensitive annotations on types, rather than a single object instance. Moreover, RegVault provides two compiler options to enable return address and function pointer protection.

2.4.2 Instruction Instrumentation. With annotations, RegVault first collects the marked data types. Then, RegVault traverses all IR instructions to identify all loads and stores on the marked data types. Finally, RegVault inserts data encryption instructions with proper tweaks before data stores, and data decryption instructions after data loads. For example, to protect kernel pointers, RegVault instruments pointer stores with the encryption primitives `creak` (Figure 2a Line 2) and pointer loads with the decryption primitives `crdak` (Figure 2a Line 6). In addition, for 32-bit and 64-bit data protection, the instrumented code are shown in Figure 2b and 2c respectively. Moreover, to defeat spatial substitution attacks, RegVault uses storage addresses as encryption tweaks for most data. Therefore, RegVault should deal with functions like `memcpy` properly. RegVault first identifies the copied data type by tracing the type information of the source and destination pointers, then re-encrypts the annotated fields within the copied data using the new addresses as tweaks.

2.4.3 Interrupt Context Protection. During an interrupt, kernel stores all general purpose registers into the interrupt context in memory. As a result, the unprotected interrupt context gives the attacker chances to leak and corrupt these register values [2]. Therefore, RegVault proposes a novel technique named *chain-based interrupt context protection* (short for *CIP*), which not only encrypts the interrupt context but also protects its integrity.

The basic idea of CIP is to form a chain-based de-randomization process, so that the next register value is decrypted using the previous decrypted register value as the tweak, as shown in Figure 4. Moreover, CIP inserts a zero value at the end of encryption and decryption for integrity checking. In this way, CIP can detect any corrupted register values in the middle of interrupt context, by verifying whether the decrypted ending zero remains zero. Moreover,

Table 2: Protected kernel data in RegVault.

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

to enhance the security of CIP, we 1) use the storing address as the tweak for the first register encryption to defeat spacial substitution attacks; 2) use a dedicated key register for CIP to defeat cross-data-type substitution attacks, and 3) maintain different interrupt keys for different threads to thwart cross-thread substitution attacks.

2.4.4 Sensitive Register Spilling Protection. Sensitive data might be spilled to memory in plaintext due to 1) the lack of physical registers and 2) function calls. Therefore, to achieve comprehensive protection, RegVault must handle these spillings securely.

Identifying Sensitive Registers: The annotated sensitive data must be decrypted or encrypted whenever it enters or leaves registers. Therefore, RegVault marks the following registers as sensitive: 1) the plaintext registers in RegVault cryptographic operations and 2) the registers propagated from or to other sensitive registers.

Intra-procedural Spilling Protection: RegVault increases the cost of sensitive register spilling so that they are less likely to be spilled. And if a sensitive register has to be spilled, RegVault inserts cryptographic primitives around the store and reload instructions.

Inter-procedural Spilling Protection: To protect spilling during function calls, RegVault proposes a novel *cross-call spilling protection*. For sensitive caller-saved registers, i.e., spilled by the caller, RegVault identifies their spilling and inserts cryptographic primitives to protect them. For sensitive callee-saved registers at call-sites, i.e., spilled by the callee, RegVault inserts cryptographic primitives to ensure they are encrypted when entering the callee.

3 REGVAULT APPLICATION

RegVault 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. For the non-control data, without loss of generality, we apply RegVault to protect encryption keys, the user credentials, security feature switches and memory management data. With the annotation support, RegVault can be used to protect more kernel data easily.

3.1 Control Data

3.1.1 Return Address. RegVault inserts `creak ra, ra[7:0], sp` into the prologue to encrypt the return address, and `crdak ra, ra, sp, [7:0]` into the epilogue to decrypt the return address. The stack pointer `sp` is used as the tweak to diversify the randomization. Moreover, RegVault adds a per thread key field to the `thread_info`, which is initialized at thread forking, encrypted by the master key in memory and written to key register on context switches. Therefore, each thread's return addresses are encrypted with a unique key.

3.1.2 Function Pointer. RegVault detects the function pointer usages and inserts the encryption and decryption instructions automatically. The instrument code is similar to the one in Figure 2a. RegVault uses a dedicated key to randomize kernel function pointers only. Moreover, the storage addresses are used as the tweaks to diversify the randomization. To identify function pointers, RegVault adopts an over-approximate approach by identifying the function pointer type and regarding all `void *` pointers as function pointers.

3.2 Non-control Data

3.2.1 Kernel Keys. Linux kernel provides kernel keyrings to store cryptographic keys. Unfortunately, these keys are stored as the plain text and are vulnerable to memory disclosures. We use RegVault cryptographic primitives to protect their confidentiality. RegVault ensures that cryptographic keys are always encrypted in memory. To achieve this, during key setup phases, RegVault encrypts these keys before storing them. After that, in encryption and decryption functions, RegVault inserts decryption instructions immediately after loading the keys. As a proof of concept, we apply RegVault to the AES engine in Linux kernel `crypto` subsystem and RegVault successfully prevents AES key disclosure attacks, as detailed in §4.3.1.

3.2.2 User Credentials. The attacker often corrupts the `uid/gid` fields of `cred` struct to escalate their privileges [21]. To defeat these attacks, RegVault randomizes `uid/gid` with integrity protection, as discussed in §2.4.2.

3.2.3 Security Feature Switches. Linux security features usually use kernel variables to control their behaviors. For example, the `selinux_enforcing` and `ss_initialized` control the on/off of SELinux. Unfortunately, researchers have demonstrated attacks that overwrite these variables to disable SELinux [22]. In recent Linux kernel, these variables are gathered in a global struct named `selinux_state`. However, the code logic remains the same, leaving the same weak spot. To make SELinux more robust against vulnerabilities, RegVault randomizes all fields inside `selinux_state` (except the lock fields) with integrity protection.

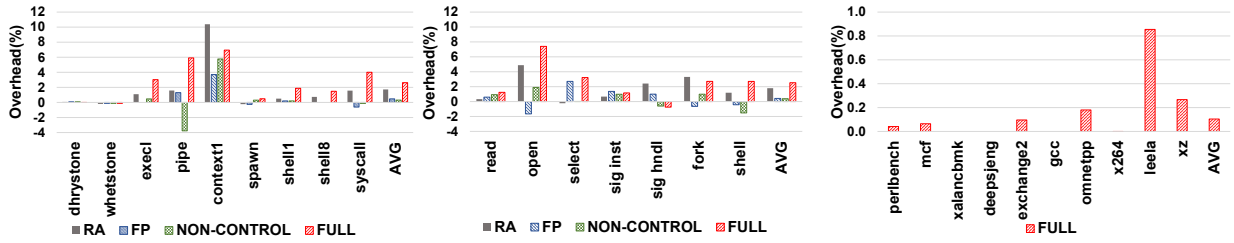
3.2.4 PGD Pointers. Page tables are critical kernel data that manage memory access permissions. Unfortunately, they are globally writable in the kernel space, allowing the attacker to disable memory protection [15]. Therefore, RegVault proposes to randomize every PGD pointer to hide page table locations. RegVault instruments all loads and stores of PGD pointers, by annotating the `pgd_t` type. To defeat substitution, RegVault uses the storage addresses of PGD pointers as the tweaks to diversify the randomization. Moreover, to prevent the attacker from locating page tables allocated statically, RegVault re-allocates these page tables and page table entries, and updates all their references.

4 IMPLEMENTATION AND EVALUATION

In this section, we present the implementation of RegVault along with its security analysis and performance evaluation.

4.1 Implementation

We implement the QARMA-based crypto-engine and the decoding unit for cryptographic instructions with about 820 lines of code changes in Chisel for Rocket core. For the software support, we apply RegVault to Linux kernel v5.8.18 with 750 lines of code changes. RegVault also extends Clang/LLVM 11 with about 4000 lines of code changes.



(a) UnixBench results. The average overhead for full protection is 2.6%.

(b) LMbench results. The average overhead for full protection is 2.5%.

(c) SPEC2017 intspeed results. The average overhead for full protection is close to zero.

Figure 5: Micro-benchmark and macro-benchmark performance.

Table 3: RegVault relative hardware resource cost over the entire SoC, compared with FPU.

CLB Num		RegVault Crypto-Engine %	RegVault CLB %	FPU %
0	#LUT	4.88	N/A	25.28
	#FF	4.79	N/A	12.40
8	#LUT	4.42	4.30	24.39
	#FF	4.55	4.84	11.78

Table 4: Penetration test results. RegVault stops all attacks.

Attacks	Defenses	
	Original	RegVault
❶Return-Oriented Programming	✗	✓
❷Jump-Oriented Programming	✗	✓
❸Sensitive Data Corruption	✗	✓
❹Sensitive Data Leak	✗	✓
❺Privilege Escalation	✗	✓
❻SELinux Bypass[22]	✗	✓
❼Interrupt Context Corruption	✗	✓
❽Spatial Code Pointer Substitution	✗	✓

4.2 Hardware Evaluation

We implement the RegVault prototype with 8 CLB entries on a Xilinx Virtex-7 VC707 board with 1GB DDR3 memory running on 100 MHz. Our implementation of the crypto-engine completes the QARMA cipher in 3 cycles and does not fall into any timing critical path. We evaluate the relative FPGA hardware resources utilized by RegVault with different numbers of CLB entries, such as the look-up tables (LUTs) and the flip-flops (FFs). The results are shown in Table 3. Both RegVault crypto-engine and CLB have low hardware resource cost (<5% over the SoC). Compared to the floating-point unit (FPU), RegVault consumes a relative low portion of hardware resources.

4.3 Security Analysis

To evaluate the security of RegVault thoroughly, we first perform penetration tests on RegVault using real-world attacks, and then discuss a RegVault specific attack.

4.3.1 Penetration tests. We port the RIPE attack suite [23] to the RISC-V Linux kernel, and simulate ❶ROP, ❷JOP, ❸sensitive data corruption attacks, and ❹sensitive data disclosure attacks. Besides, we also develop real-world attacks against RegVault, including ❺privilege escalation by corrupting `cred.uid`, ❻SELinux bypass by corrupting `selinux_state.initialized` [22], ❼interrupt context corruption by tampering a register saved in interrupt context, and ❽spatial code pointer substitution by replacing an encrypted function pointer by another in a different address.

As shown in Table 4, RegVault can defeat all of the above attacks. More specifically, the 64-bit randomization makes unauthorized writes unpredictable (defeating ❶,❷,❸) and prevents data leak (defeating ❹). The integrity protection blocks general data corruption (defeating ❺,❻). And the address-based randomization thwarts spatial substitution attacks (defeating ❸). Moreover, interrupt context corruption is detected by our chain-based interrupt context protection (defeating ❼). Therefore, RegVault is capable of stopping state-of-the-art attacks on protected sensitive data.

4.3.2 Time-of-derandomize-to-time-of-use attack. In RegVault, the data de-randomization and the use are not in an atomic instruction, giving the attacker chances to exploit the gap and corrupt or leak sensitive data. To address this problem, we propose to randomize the interrupt context with integrity protection, as detailed in §2.4.3. As a result, the confidentiality and the integrity of interrupt context are protected by RegVault, defeating any attempts to leak or corrupt sensitive data.

4.4 Performance Evaluation

4.4.1 CLB Performance. We collect the run-time information when running UnixBench, and calculate hit ratios under different numbers of CLB entries. The result shows that a CLB with only 8 entries can achieve 51.7% hit ratio, indicating most decryption instructions can find the corresponding plaintext result from the CLB directly. The results from the UnixBench show that the CLB decreases the full protection overhead by 1.9%, from 4.5% to 2.6%.

4.4.2 Micro-benchmark Performance. We select UnixBench and LMbench for micro-evaluation. As shown in Table 2, we test the performance overhead with four protection configurations: protecting return addresses only (RA), protecting function pointers only (FP), protecting four types of non-control data (NON-CONTROL), and full protection including all of the above data (FULL). The full protection also includes the interrupt context and register spilling protection. As shown in Figure 5a and 5b, enabling all protection, RegVault achieves a 2.6% overhead for UnixBench, and a 2.5% overhead for LMbench. As these micro-benchmarks are syscall-oriented, their results provide upper bounds of performance overhead for userspace programs.

4.4.3 Macro-benchmark Performance. We use the intspeed test suite of SPEC CPU2017 as the macro-benchmark. As shown in Figure 5c, the performance overhead is a close-to-zero for the full protection, indicating that RegVault has the minimal performance impact on common userspace programs.

5 RELATED WORK

Data Space Randomization (DSR). Data space randomization [6] leverages pointer analysis to partition the data into different equivalence classes, and assigns a random XOR mask to each class. To protect the secret XOR masks, HARD [5] achieves context-aware data partition and stores the masks in a protected memory key table. Moreover, CoDaRR [19] further proposes to re-randomize the masks periodically. Nevertheless, all of these works suffer memory disclosures, due to the weak XOR-based encryption.

Selective Data Protection. For code pointer protection, CPI [11] allocates an isolated safe memory region for code pointers and all data pointers used to access code pointers. CCFI [14] uses AES-NI instructions to generate message authentication codes (MACs) for code pointers, and verifies their integrity based on the MACs. For general selective data confidentiality protection, Palit et al. leverage static analysis to track sensitive data flow and insert AES operations to keep selected sensitive data encrypted in memory [16]. DynPTA [17] further combines static analysis with dynamic data flow tracing to reduce the performance overhead. However, both of them incur high runtime overhead (more than 10%). Compared to them, RegVault protects the confidentiality and integrity for both code pointers and general data, with a minimal performance overhead.

Hardware Cryptographic Primitives. Researchers have developed various lightweight ciphers optimized for performance in hardware implementations. Traditional block ciphers like Simon [3] do not accept an extra input. Therefore, they suffer substitution attacks and cannot enforce strong integrity. Other lightweight tweakable block ciphers, like CRAFT [4], are compatible with RegVault architecture. RegVault can provide similar security guarantee based on these ciphers. Standalone cryptographic processors like Cryptoraptor [20] support a wide range of cryptographic algorithms with high throughput. But the high communication latency between cryptographic processors and CPU makes them unsuitable for selective register-grained data protection. Morpheus [10] is a hardware randomization architecture. But it adds context information to encryption using weak XOR instead of the tweak and does not safeguard general data.

ARMv8.3 introduces Pointer Authentication (PA) [18], which is a hardware-assisted mechanism to sign and authenticate pointers. Leveraging PA, researchers have developed context sensitive protection on pointers [9, 12, 13]. Nevertheless, ARM PA is designed for enforcing integrity, and thus cannot provide confidentiality protection for general data. In contrast, RegVault hardware primitives introduce flexible range selection, which can achieve both the confidentiality and the integrity protection.

6 CONCLUSION

This paper presents RegVault, a hardware-assisted selective data randomization scheme for OS kernels. RegVault leverages new hardware primitives to protect both the confidentiality and integrity of register-grained data. RegVault uses annotations to mark sensitive data and instruments their loads and stores automatically. We implement a prototype of RegVault by extending RISC-V architecture to protect six types of kernel sensitive data. The evaluations show that RegVault can defend against the kernel data attacks effectively with a minimal performance overhead.

REFERENCES

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

[2] Brandon Azad. ios kernel pac, one year later. https://i.blackhat.com/USA-20/Wednesday/us-20-Azad-iOS-Kernel-PAC-One-Year-Later.pdf, 2020.

[3] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. https://ia.cr/2013/404.

[4] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. Craft: Lightweight tweakable block cipher with efficient protection against dfa attacks. Cryptology ePrint Archive, Report 2019/210, 2019. https://ia.cr/2019/210.

[5] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 337–358. Springer, 2018.

[6] Sandeep Bhatkar and R Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[7] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the ASIACCS*, pages 193–204, 2017.

[8] Microsoft Corporation. A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[9] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted cfi for the arm linux kernel. In *Proceedings of DAC*. IEEE Press, 2020.

[10] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, et al. Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn. In *Proceedings of the ASPLOS*, pages 469–484, 2019.

[11] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

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

[13] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[14] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[15] Enrique E. Nissim Nicolas A. Economou. Getting physical: extreme abuse of intel based paging systems. https://www.coresecurity.com/core-labs/articles/getting-physical-extreme-abuse-of-intel-based-paging-systems, 2016.

[16] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. Mitigating data leakage by protecting memory-resident sensitive data. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 598–611, 2019.

[17] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[18] Inc. Qualcomm Technologies. Pointer authentication on armv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[19] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Codarr: Continuous data space randomization against data-only attacks. In *Proceedings of ASIACCS*, pages 494–505, 2020.

[20] Gokhan Sayilar and Derek Chiou. Cryptoraptor: High throughput reconfigurable cryptographic processor. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 155–161, 2014.

[21] INetCop Security. New reliable android kernel root exploitation techniques. http://powerofcommunity.net/poc2016/x82.pdf.

[22] Di Shen. Defeating samsung knox with zero privilege. https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege.pdf.

[23] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. Ripe: Runtime intrusion prevention evaluator. In *ACSAC*, 2011.