



SDP: Sensitive Data Protection

Zhiyao Feng[†], Anjo Vahldiek-Oberwagner[‡], Hans Liljestrand^{§1}, Atri Bhattacharyya[†],
Thomas Bourgeat[†], Mathias Payer[†]
[†]EPFL [‡]Intel [§]Huawei

Abstract—System software implemented in low-level languages like C remains prone to memory safety vulnerabilities. These vulnerabilities may be exploited by attackers to corrupt data and compromise the system. Among all data in program memory, a small fraction is highly security-sensitive, e.g., critical system or process attributes that attackers must corrupt to achieve malicious goals, and therefore becomes the ultimate attack target. Attackers corrupt such data through data-only attacks or control-flow hijacking. Existing solutions defend against all attack paths by stacking multiple mitigations for different program assets to protect, leading to increased complexity, compounded overhead, and inherited security weaknesses of each mitigation.

SDP introduces a unified solution to protect sensitive data against both data-only attacks and control-flow hijacking. We convert the sensitive data protection problem into a sensitive pointer protection problem, and propose a hardware-assisted mechanism, pointer type-based compartmentalization, to enforce protection on sensitive pointers against unauthorized access. We prototype SDP extending the RISC-V ISA, evaluate its security guarantees by protecting 16 commonly attacked security-sensitive fields in the Linux kernel, and show its negligible performance impact with a set of micro (LMbench) and macro (Phoronix) benchmarks.

1. Introduction

System software like OS kernels is complex, of diverse provenance, and often implemented in low-level languages such as C. While such languages combine performance with fine-grained control over memory and hardware, they lack memory safety guarantees. New code is increasingly written in safe languages like Rust, but the majority of system software remains in C, and the transition to fully safe code will take decades.

Memory safety vulnerabilities like buffer overflows or use-after-free are still prevalent. Attackers can exploit them to gain restricted or arbitrary read and write capabilities [1], to locate and corrupt data in memory and compromise the system. Large language models are lowering the barrier to automated vulnerability discovery and exploitation [2], [3], which exacerbates the threats.

Among all data in program memory, a small fraction is highly security-sensitive. Semantically, such data describes

critical system or process attributes, so attackers must corrupt it to achieve various malicious goals. For example, in Linux, they may manipulate system management data (e.g., page tables) to break resource access rules, forge critical process attributes (e.g., `.cred`, `.nsproxy`, `.fs` in `struct task_struct`) to escalate privileges and escape containers, clobber CPU metadata (e.g., `struct thread_struct`) to pivot execution, alter security mechanism metadata (e.g., `struct selinux_state`) to bypass or weaken the protection, tamper with audit control data (e.g., `aa_g_audit` in AppArmor) to avoid leaving traces of attack activities, or flip feature toggles (e.g., `sysctl_io_uring_disabled`, `panic_on_oops`) to expand the attack surface and facilitate further exploitation.

Memory safety violation mitigations aim to protect all data in memory, including sensitive data. However, with the focus on all data, such mechanisms may introduce significant overhead (e.g., CHERI [4], [5]), or provide less comprehensive protection (e.g., ARM MTE [6]).

Since sensitive data is the ultimate target that attackers must corrupt to achieve their malicious goals, protecting it can effectively stop the attack chain at its most critical step. Therefore, protecting this small fraction of data is a more efficient and practical approach to stop attacks compared to protecting all data in memory. We focus on sensitive data protection in this paper, aiming to provide strong security guarantees with minimal performance overhead.

With the capabilities to read and write certain data in memory, attackers can corrupt sensitive data through data-only attacks or control-flow hijacking [1]. Through data-only attacks, attackers directly overwrite the target *sensitive data*, or overwrite *sensitive data pointers* (pointers used to access sensitive data) to redirect them to locations with desired values. Through control-flow hijacking, attackers overwrite *code pointers* or *indirect code pointers* (pointers used to access code pointers), and redirect execution to the code path that sets sensitive data or pointers.

Existing mitigations stack multiple mechanisms to protect these program assets, which increases system complexity and compounds performance overhead. The composed defense also inherits the security weaknesses of every mechanism, while attackers need to break only one of them. For example, xMP [7] isolates sensitive data and protects sensitive data pointers with HMACs. PeTAL [8] protects access control-related data with ARM MTE [6] and pointers with ARM Pointer Authentication (PA) [9]. They both require

1. Part of this work was done while at Intel.

integration with Control-Flow Integrity (CFI) [10], [11] to be effective, and are exposed to reuse [12] or brute-force attacks targeting HMACs, PACs, and MTE tags.

In this paper, we propose SDP, *a unified solution to protect sensitive data from unauthorized corruption through both data-only attacks and control-flow hijacking*.

We protect the four kinds of program assets with the following policy. For *sensitive data*, we hide it to prevent attackers from locating and corrupting it. To hide sensitive data, we randomly allocate it at unpredictable addresses in separate heap partitions within a large virtual address space, and restrict attackers from unauthorizedly reading sensitive data pointers. We also randomly set tripwires around each sensitive data object to thwart random guessing or brute-force attacks that attempt to locate sensitive data. For *sensitive data pointers*, *code pointers*, and *indirect code pointers*, which we call *sensitive pointers*, we restrict attackers from unauthorizedly writing them.

This way, we convert the problem of protecting sensitive data against data-only attacks and control-flow hijacking into protecting sensitive pointers against unauthorized read and write operations, which simplifies the overall system design, avoids compounded overhead and security weaknesses, and allows for an effective yet efficient and concise policy enforcement mechanism.

We employ hardware-assisted pointer type-based compartmentalization to protect sensitive pointers from unauthorized access. During compilation, we identify the `load` and `store` instructions that are generated for accessing sensitive pointers and separate them into distinct *sensitive compartments* for each sensitive pointer type (e.g., `struct cred*`). The `load` and `store` instructions that are generated for accessing other types of data, i.e., normal and sensitive data, fall into the *default compartment*. At runtime, with our hardware extension, we enforce that the `load` and `store` instructions in each compartment cannot access objects in other compartments. This constrains any exploited vulnerability to a single compartment, i.e., the compartment that contains the attacker-controlled vulnerable `load` or `store` instruction. If the vulnerability falls into the default compartment, which is the common case according to probabilistic reasoning and our empirical analysis of KernelCTF exploits [13], attackers cannot unauthorizedly access any sensitive pointers. In the rare case where it falls into a sensitive compartment, attackers can only unauthorizedly access sensitive pointers of the same type, which grants attackers minimal benefits and limits the potential damage.

We prototype SDP extending the RISC-V ISA [14] and build (1) a QEMU-based [15] functional emulator to evaluate SDP’s security guarantees, and (2) a performance model that executes on a RISC-V development board [16] modeling SDP instructions with performance-equivalent instructions from the existing ISA to evaluate SDP’s performance impact. We protect 16 commonly attacked security-sensitive data fields, return addresses, and function pointers in the Linux kernel, reproduce exploits targeting them, and show that SDP can effectively mitigate them all. SDP imposes negligible performance impact: protecting the 16 sensitive

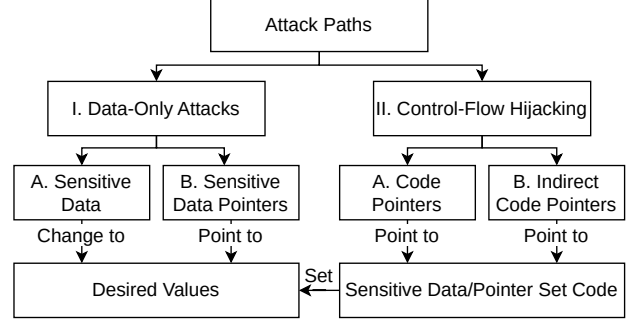


Figure 1. Attack paths to sensitive data corruption.

fields against data-only attacks, control-flow hijacking, and both, SDP averages $0.98\times$, $0.95\times$, and $0.94\times$ relative to the baseline on standard kernel microbenchmarks in LM-bench [17], and $1.00\times$, $1.01\times$, and $0.99\times$ on stress tests and real-world user space applications in Phoronix [18].

This paper makes the following contributions:

- We convert the sensitive data protection problem into a sensitive pointer protection problem, and propose SDP to protect sensitive data against both data-only attacks and control-flow hijacking in a unified solution.
- We propose a hardware-assisted pointer type-based compartmentalization mechanism, which protects sensitive pointers against unauthorized access.
- We demonstrate SDP’s strong security guarantees and negligible performance impact with a prototype implementation and evaluation.

The SDP prototype is publicly available at <https://github.com/HexHive/SDP>.

2. Background

Through vulnerability exploitation, attackers can read or write certain regions in the process’s address space [1]. Starting from these basic primitives, they progressively expand their capabilities, with the ultimate goal of corrupting sensitive data and compromising the system. We show the attack paths for sensitive data corruption in Figure 1.

I. Data-only attacks. *Attack Path I.A: Locate and corrupt the sensitive data itself.* If the sensitive data is reachable, attackers can directly locate and overwrite it with crafted values, e.g., overwriting the credential ID fields in `struct cred` with 0 to escalate to root privileges.

Attack Path I.B: Locate and corrupt sensitive data pointers. Sensitive data is often accessed indirectly through pointers. Attackers can overwrite the direct pointer to sensitive data, and redirect it to a location holding the desired values. The target location can be an attacker-controlled buffer with a fake sensitive data structure, or an existing data structure with the desired values. For example, instead of overwriting `struct cred`, attackers can redirect its pointer in `struct task_struct` to `init_cred` to escalate privileges. Moreover, some sensitive data is accessed by dereferencing a chain of pointers. Attackers can overwrite

any indirect pointer in the chain, and redirect the rest of the chain to a new location with desired values.

II. Control-flow hijacking. *Attack Path II.A: Locate and corrupt code pointers.* Attackers can overwrite a code pointer (a function pointer or return address), to redirect execution to the code path that sets sensitive data or pointers, e.g., `commit_creds()` that installs new credentials upon the current task. Then, the target sensitive data or pointer gets overwritten as the execution continues.

Attack Path II.B: Locate and corrupt indirect code pointers. As with sensitive data, some code pointers are accessed indirectly through data pointers, e.g., `.read` accessed via `struct file.f_op` in `vfs_read()`. Attackers can overwrite these data pointers and redirect them to a new location containing the desired code pointer.

In summary, to corrupt sensitive data, attackers can overwrite the sensitive data itself, direct and indirect sensitive data pointers, and direct and indirect code pointers. We consider protecting these program assets effectively and efficiently in SDP.

3. Threat Model

Target programs. SDP is designed to protect security-sensitive data in memory for 64-bit C programs with a 48-bit virtual address space, particularly for OS kernels such as the Linux kernel. We assume that the target program contains at least one memory safety vulnerability, e.g., buffer overflows or use-after-free. SDP can be configured to mitigate sensitive data corruption through data-only attacks, control-flow hijacking, or both. When control-flow hijacking is not mitigated by SDP, the target programs should be equipped with mitigations like CFI. We assume code sections are not writable, and data sections are not executable, so no code modification or injection attacks are possible.

Attacker capabilities and goals. We assume powerful yet realistic attacker capabilities. Attackers have access to the source code and binary image of the target program, as well as arbitrary read and write capabilities through vulnerability exploitation. When applying SDP to protect OS kernels, we assume that the attacker is initially unprivileged. The attack goal is to corrupt security-sensitive data and compromise the system, e.g., escalating privileges, escaping containers, or bypassing security mechanisms, rather than launching Denial-of-Service attacks.

Trusted computing base. We assume that the compiler, linker, loader, and hardware are benign and functioning correctly. We consider the physical environment part of the trusted computing base, so attacks like cold boot [19] are out of scope. Microarchitectural side-channel attacks (e.g., speculative execution attacks [20]) are out of scope and deferred to orthogonal defenses.

4. SDP Policy

SDP protects sensitive data, direct and indirect sensitive data pointers, and direct and indirect code pointers to miti-

gate sensitive data corruption through data-only attacks and control-flow hijacking.

Core idea. The program assets to protect fall into two categories: sensitive data and sensitive pointers. They play different roles in programs and have different characteristics, calling for tailored protection mechanisms. Applying tailored write protection on each category to restrict unauthorized write operations from attackers, as in xMP [7] and PeTAL [8] against data-only attacks, is straightforward, but compounds complexity, overhead, and security weaknesses.

In SDP, we propose to: (1) hide sensitive data, to prevent attackers from locating and corrupting sensitive data directly (*Attack Path I.A*), and (2) apply write protection on sensitive pointers, to restrict attackers from redirecting access to sensitive data (*Attack Path I.B*) or control flow (*Attack Path II.A and II.B*), and corrupting sensitive data indirectly.

Hiding sensitive data. Attackers attempt to locate sensitive data in one of three ways: (1) leaking direct sensitive data pointers, (2) inferring the relative offset of sensitive data from a known data structure, or (3) random guessing.

To mitigate the above locating attacks and hide sensitive data effectively, we (1) apply read protection on direct sensitive data pointers (e.g., `struct task_struct.cred`), to restrict their leakage, and (2) randomly allocate sensitive data (e.g., `struct cred`) within a large virtual address space, to mitigate relative offset-based locating and random guessing. Specifically, we allocate each type of sensitive data in a separate heap partition, randomly choose the partition base address at runtime, and set random tripwires around each allocated data object. Then the sensitive data addresses become unpredictable to attackers, and the probability of successful random guessing is negligible in practice.

Unifying pointer protection. Direct sensitive data pointers require both read and write protection, while the other sensitive pointers require only write protection. Instead of having two protection levels, we apply read and write protection on all sensitive pointers. It still provides strong security guarantees, while reducing the implementation complexity and metadata overhead (1 bit to distinguish between the two protection levels) for the policy enforcement mechanism.

SDP policy. (1) Randomly allocate sensitive data. (2) Apply read and write protection on all sensitive pointers to restrict unauthorized access from attackers.

Benefits. SDP can mitigate all attack paths to sensitive data corruption with sufficient security guarantees in practice. We discuss them in detail in Section 7.

SDP supports a wide range of use cases, even though we allocate sensitive data only in heaps. We protect sensitive pointers in heaps, on stacks, and in global sections. Heap sensitive data is most common and can be protected by SDP naturally. Global sensitive data can be protected in an indirect way, by allocating the actual data in heaps and keeping its pointers in the global section. Stack sensitive data is rare in practice, but can be protected in the same way as global sensitive data if needed.

Moreover, SDP provides a unified approach for sensitive data protection, instead of stacking three separate

mechanisms to protect sensitive data, sensitive pointers, and mitigate control-flow hijacking like existing solutions [7], [8]. The unified approach also leads to reduced overhead and a concise policy enforcement mechanism as follows.

5. Pointer Type-Based Compartmentalization

In this section, we describe the sensitive pointer protection mechanism for SDP policy enforcement.

Distinguishing unauthorized access. To provide read and write protection for sensitive pointers, we must distinguish between authorized and unauthorized access at runtime and restrict the latter, but `load` and `store` instructions in executable binaries carry no such information.

When writing source code, program developers define variable types and operate on instances of them. The compiler transforms these high-level operations into low-level instructions based on variable types, then discards type information, as it is not needed for execution. However, type information captures the developer’s intention: it indicates which `load` and `store` instructions are generated to access data of specific types, and are therefore authorized to do so.

We propagate the type information of sensitive pointers from source code to their corresponding `load` and `store` instructions at compile time, and utilize it to distinguish unauthorized access to sensitive pointers at runtime.

Core idea. With the propagated type information, we apply pointer type-based compartmentalization, to create compartments of individual sensitive pointer types, and enforce that sensitive pointers can only be accessed by `load/store` within the compartment.

Definitions. We describe our mechanism following the nomenclature in the recent compartmentalization SoK [21]. In our context, a *subject* is a `load` or `store` instruction. An *object* is one or more memory bytes that a subject may attempt to access. The *permissions* are actions that subjects may perform on objects: read by `load` and write by `store`.

During compilation, we propagate type information from source code to the `load/store` instructions generated for accessing sensitive pointers. Our *compartmentalization policy* is that a `load/store` attached with type information has permissions to access only sensitive pointers of that type, while other `load/store` instructions have permissions to access only other data, i.e., normal and sensitive data.

This creates multiple *sensitive compartments*, each containing all `load/store` that can only access sensitive pointers of a single type, plus one *default compartment*, containing all `load/store` that can only access other data.

We use a 1-bit *Sensitivity Tag*, abbreviated as *STag*, to distinguish sensitive compartments (STag 1) from the default compartment (STag 0). We encode the type information into a 16-bit *TypeID Tag*, abbreviated as *TTag*, to distinguish different sensitive compartments. The STag and TTag together constitute the *compartment identifier*, abbreviated as *CID*. The default compartment has only STag 0; sensitive compartments have STag 1 plus a TTag.

Compile-time compartment determination. To determine compartments, we attach the same CID to the sub-

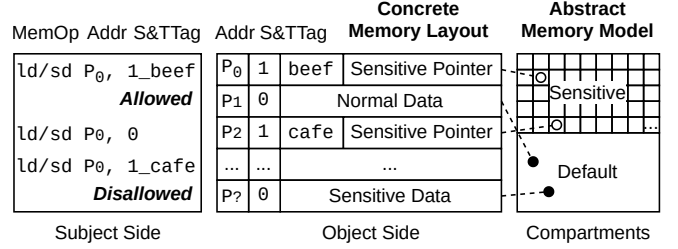


Figure 2. The pointer type-based compartmentalization mechanism in SDP. `load` and `store` instructions are allowed to proceed if and only if subject-side (in instructions) and object-side CIDs (in memory) match.

jects (`load/store` instructions) and the objects (memory bytes) they are supposed to access. We do this statically by deciding CIDs for both subjects and objects at compile time.

Subject side, we identify the `load/store` instructions that are generated for accessing sensitive pointers with type analysis. Then we encode a CID (STag 1, a TTag hashed from the sensitive pointer type) into the instruction machine code. The remaining `load/store` instructions that access other data receive a default CID (STag 0, no TTag).

Object side, we instrument dedicated instructions to attach and detach CIDs to the memory bytes of sensitive pointers at the start and end of their lifetime, so that each CID is correctly associated with the memory bytes while they hold a valid sensitive pointer and can be checked at access. Other data retains the default CID. With control-flow hijacking mitigated, attackers cannot arbitrarily redirect the compiler-defined control flow to the CID maintenance instructions to tamper with object-side CIDs.

Runtime compartmentalization policy enforcement. We extend the hardware to preserve metadata and enforce SDP compartmentalization policy.

Subject-side CIDs are encoded in instruction machine code and cannot be modified by attackers. We extend the hardware to preserve object-side CIDs and protect them from attackers. We store STags in one extended bit per eight bytes in cache and memory, and TTags in the unused significant bits of sensitive pointers. Neither is accessible to the `load/store` instructions attackers may control, which keeps the metadata safe from corruption through them.

As shown in Figure 2, during the execution of `load` and `store` instructions, SDP checks if the subject-side and object-side CIDs match to detect violations of the compartmentalization policy. When a mismatch is detected, the hardware blocks the access and raises an exception.

Benefits. Our mechanism provides several benefits over encryption- or authentication-based pointer protection mechanisms [9], [12], [22], [23]. Security wise, SDP is not susceptible to reuse attacks [12], in which attackers copy a valid pointer to another location to bypass the protection mechanism. Unlike ciphertexts and PACs that travel with the pointer value, object-side CIDs are bound to memory locations and inaccessible to `load/store` instructions, so a copied pointer does not inherit the original’s validity. Computation requirement wise, our CID checking is lightweight,

TABLE 1. SDP INSTRUCTION SEMANTICS. THE STAG AND TTAG IN THE TABLE REFER TO OBJECT-SIDE STAG AND TTAG, UNLESS NOTED. INSTRUCTIONS FOLLOW THE RISC-V I AND S INSTRUCTION FORMATS.

Instruction	Operands	Semantics
Original load	rd, imm(rs1)	if STag == 0: load/store;
Original store	rs2, imm(rs1)	else: exception.
ld_sc(d)p	rd, (rs1)	if STag == 1 and TTag == subject-side
sd_sc(d)p	rs2, (rs1)	TTag bits in (opcode, funct3, imm):
		load/store pointer bits; else: exception.
set_tag1	rs2, imm(rs1)	if STag == 0: set STag 1 and TTag (rs2);
		else: exception.
set_tag0	rs2, imm(rs1)	if STag == 1 and TTag == rs2:
		zero STag and TTag; else: exception.
clear_tag	imm(rs1)	Privileged. if STag == 1:
		zero STag and TTag; else: nop.
get_tag	rd, imm(rs1)	Privileged. return STag and TTag in rd.

requiring only simple comparison operations during memory access, whereas encryption- or authentication-based mechanisms require expensive cryptographic computations. Functionality wise, SDP does not require key management or complex cryptographic operations, which reduces attack surface, simplifies implementation, and saves energy.

6. System Design and Support

SDP protects sensitive data from data-only attacks and control-flow hijacking in a unified solution through hardware and software co-design. We now discuss the design considerations and implementation details for each component.

6.1. Architecture Support

We prototype SDP extending the RISC-V ISA [14] (Table 1). We add CID checking logic to original memory access instructions and introduce new instructions for sensitive pointer access and object-side CID maintenance.

Original memory access instructions. We extend the semantics of original `load/store` instructions at different granularities to support CID checking. We link them to the default CID (STag 0, no TTag), so they can only access memory bytes with STag 0, not sensitive pointers with STag 1. Misaligned access spanning the boundary between two aligned 8-byte granules is allowed only if both STags are 0.

Sensitive pointer access instructions. We introduce new instructions to read and write sensitive pointers. They are linked to specific CIDs (STag 1, with corresponding TTags), and can only access sensitive pointers with matching CIDs. When CIDs match, the actual memory access proceeds but only reads or writes the pointer bits, leaving TTags saved in unused significant bits untouched. Misaligned access is disallowed for these instructions.

A wider TTag reduces the risk of collisions and allows protecting more sensitive pointer types. For 64-bit C programs with a 48-bit virtual address space, we use 16-bit TTags. Object side, TTags occupy the unused bits in sensitive pointers. Subject side, we encode them into instruction machine code to enable efficient CID checking and minimal software instrumentation.

Sensitive pointer access instructions follow the RISC-V I and S instruction formats [14]. We encode the 16-bit subject-side TTag across multiple instruction fields. We use 1 bit in `opcode` to distinguish sensitive code pointers from sensitive data pointers, and introduce four sets of new instructions: `ld_scp/sd_scp` for sensitive code pointer read and write, and `ld_sdp/sd_sdp` for sensitive data pointers. The instructions in each set share the same semantics. We repurpose the 3-bit `funct3` and 12-bit `imm` fields to carry the remaining 15 TTag bits. Unlike prior work STAR [24] that instruments 1 tag-carrying `nop` for every 3 instructions, our encoding does not cause notable I-cache/TLB pressure that degrades performance.

CID maintenance instructions. We introduce new instructions to set, clear, and get object-side CIDs for different contexts. `set_tag1` and `set_tag0` set and clear STags and TTags at the start and end of sensitive pointers' lifetime.

With control-flow hijacking attacks mitigated, attackers cannot arbitrarily redirect control flow to execute these instructions, but they may corrupt the address operands (e.g., by controlling an offset to the base address) to unauthorizedly set or clear CIDs. To mitigate such attacks, we add the following security checks: `set_tag1` verifies if the current STag is 0 before setting it to 1; `set_tag0` verifies if the current STag is 1 and the current TTag matches the expected TTag (passed as an operand) before clearing them. These checks restrict unauthorized CID setting and clearing, unlike the weaker checks in ZeRØ [25] (Section 9).

We also add two privileged instructions. `clear_tag` clears STags and TTags unconditionally for batch CID clearing. `get_tag` retrieves the STag and TTag of a given address for OS page swapping. Both instructions lack security checks, so we restrict them to privileged mode and suggest using them carefully in limited contexts only.

Exceptions. On access violations, i.e., STag or TTag mismatch, the hardware raises an exception to the exception handler once the instruction becomes non-speculative.

6.2. Microarchitecture Design

We now discuss object-side CID storage, and support for memory access and CID maintenance instructions.

Object-side CID storage. We store STags in an extra bit per eight bytes in cache and main memory. In cache, we extend eight bits per 64-byte cache line to store STags alongside data for efficient checking. In main memory, we store STags in spare ECC bits, as in SPARC ADI [26], [27], Califorms [28], ZeRØ [25], and [29]. With techniques like Combining Tags with Error Codes [30] and Implicit Memory Tagging [31], we can preserve ECC functionality while storing STags. This approach allows STags to be fetched together with data without additional memory access.

We store TTags in sensitive pointer unused bits in cache and main memory, requiring no extra storage. Importantly, although TTags reside in sensitive pointers, memory access instructions that attackers may control cannot access them: the STag 1 blocks original `load/store`, and only allows `ld/sd_sc(d)p` to access pointer bits, but not TTag bits.

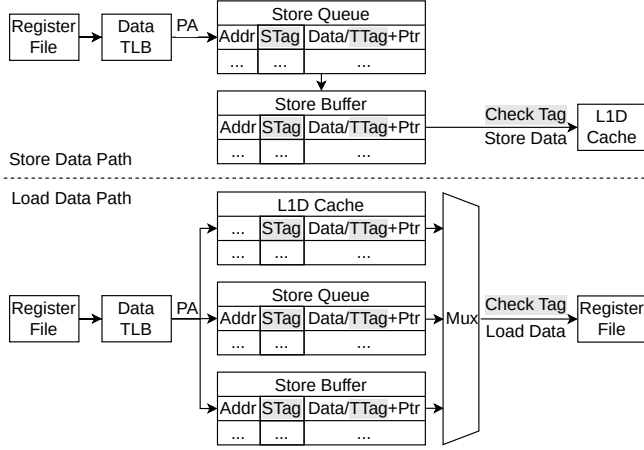


Figure 3. Store and load data paths in SDP.

Memory access support. We add CID checking logic to the load and store data paths for memory access instructions, as shown in Figure 3.

For store instructions, we pass subject-side CIDs from the instruction decoding stage to the store queue and store buffer. We extend one bit for each entry in the store queue and store buffer to keep subject-side STags. For original store, we set this bit to 0. For `sd_scp` and `sd_sdp`, we set it to 1, and save the subject-side TTag in the pointer’s unused bits within the entry’s data field, at no extra storage cost. Object-side CIDs are obtained from the L1 data cache.

For load instructions, we pass subject-side CIDs from the instruction decoding stage to the register file write-back stage. Modern processors support store-to-load forwarding for optimization, where `load` directly obtains data from store queue or store buffer entries. In such cases, object-side CIDs are obtained from these entries; otherwise, they are obtained from the L1 data cache.

In both store and load data paths, we check subject- and object-side CIDs in parallel with existing operations within the same pipeline stage. In the store data path, we check CIDs when writing data from the store buffer to the L1 data cache. In the load data path, we check CIDs when writing data back to the register file. Any CID mismatch triggers an exception and blocks the access.

In the case of store-to-load forwarding, we cannot confirm if the forwarded object-side CID is legal, as the source store has not been checked. So an attacker might execute a store with subject-side CID A to write memory bytes with object-side CID B, then forward the value to a load with subject-side CID A. This does not compromise security, as the leaked value originates from the attacker, and an exception is raised once the store’s CID checking is done.

Our CID checking does not introduce new speculative execution attack vectors. When an unauthorized load is speculatively executed, the check fails and causes an access violation. We enforce that once a violation occurs, no data is forwarded to any younger dependent speculative instruction, preventing sensitive pointer leakage through side channels.

CID maintenance support. We handle the CID maintenance instructions similarly to memory access instructions, while accessing only CID bits, not data or pointer bits. For `set_tag1` and `set_tag0` that require CID checking, we perform the check in the same way as for store instructions. For `clear_tag` and `get_tag`, we allow them to access CID bits without checking only in privileged mode.

6.3. Software Design

We now describe the required software changes from program developers, the compiler, and the operating system to support SDP.

6.3.1. Manual effort. Like related work xMP [7], SDP requires some manual effort from developers to protect sensitive data and pointers.

Sensitive data identification. We focus on sensitive data protection policy and mechanism in this paper, and consider its automatic identification as an orthogonal problem. Developers could specify sensitive data based on their semantic understanding of the system and the security requirements. We select 16 sensitive data fields in the Linux kernel for SDP evaluation and justify our selection in Section 8.1.

Sensitive data allocation. Developers create a heap partition for each type of sensitive data and allocate sensitive data objects with SDP random allocator.

Sensitive data access. For sensitive data in global sections, on stacks, or in heaps but embedded as part of other data structures, developers introduce an indirection by saving a pointer to the actual data (in the per-type heap partition) at the data’s original location, and updating the code to access the data through this pointer.

Sensitive pointer annotation. Developers annotate direct sensitive data pointers with a `__tag1` keyword. We do not infer them from the pointee type, as possibly only a subset of pointers to that type needs protection, e.g., only the `char*` pointing to sensitive strings should be protected, not every `char*`. Distinguishing these cases requires semantic knowledge from developers. For indirect sensitive data pointers, code pointers, and indirect code pointers, SDP automatically identifies them, requiring no annotations.

False positive avoidance. Like function pointer type-based CFI, e.g., KCFI [32], SDP requires code changes to avoid false positives. For example, SDP does not protect sensitive pointers within unions, as the valid member’s type is unknown at CID setting sites. Developers must move such pointers to separate data structures, convert the union into structs, or skip their protection, depending on the refactoring cost and security requirements. SDP emits memory access instructions with correct CIDs at the C level, but not for hard-coded assembly, so developers must use `ld/sd_sc(d)p` instructions with correct CIDs to access sensitive pointers there. SDP emits CID maintenance instructions assuming standard C type usage, so developers must avoid patterns that violate it, e.g., `struct scm_fp_list` in Linux declares its member `.fp` as an array of

size `SCM_MAX_FD`, but deliberately under-allocates it to the actual `fd` count, while setting CIDs for the whole array overflows into adjacent data and causes false positives.

6.3.2. Compiler support. SDP requires sensitive pointer annotation, identification, compartment determination, and bulk memory operation handling. We support these features on top of LLVM 19.1.2 [33] for the RISC-V backend.

Keywords and compilation options. We provide a `__tag1` keyword for developers to annotate direct sensitive data pointers. To mitigate control-flow hijacking, we provide compilation options that automatically identify and protect code pointers: `-fsdp-ret` for return addresses, `-fsdp-fp` for function pointers, and `-fsdp-fpifp` for direct and indirect function pointers. The automatically identified sensitive pointers may fall into false positives, e.g., as union members, and the developer may choose to skip their protection, so we also provide a `__tag0` keyword to force a pointer to be insensitive.

Sensitive pointer identification. For direct code pointers, we identify return addresses by their stack location and function pointers by their pointee type. For indirect sensitive data and code pointers, we identify them at pointer type building time by checking if the pointee type (1) is a sensitive pointer type, or (2) contains a sensitive pointer type, e.g., arrays or structs holding sensitive pointers.

Forward declarations in C yield incomplete struct types, whose pointer sensitivity cannot be determined within the compilation unit. Rather than conservatively treating all such pointers as sensitive like CPI [34], we compile the program twice to analyze them precisely and avoid unnecessary runtime overhead. The first compilation records the field sensitivity of every complete struct type and marks fields with undetermined sensitivity, e.g., a struct pointer, a struct holding undetermined fields. Then with the collected information, we recursively resolve the sensitivity of undetermined fields and update the struct types' sensitivity accordingly, until reaching a fixed point. Structs that remain undetermined, due to cyclic dependencies or third-party library definitions, are assumed to have no sensitive fields. The second compilation reads the resolved struct sensitivity and identifies indirect sensitive pointers using the two rules above.

Sensitive pointer types. We attach a sensitive attribute to all annotated or identified sensitive pointers and treat them as distinct types from the original, so that the boundary between sensitive and insensitive pointers is clear and can be maintained by the LLVM type system. Each sensitive pointer type carries a 16-bit TTag: 1 bit to distinguish code and data pointers, and 15 bits calculated by hashing the pointer type. We hash TTags rather than assigning them sequentially to ensure that matching types in separately compiled binaries have identical TTags, which enables seamless integration with third-party libraries. We discuss the risks of TTag hash collisions in Section 7.3.

Compartment determination. We statically decide both subject-side and object-side CIDs at compile time.

Subject side, we pass the sensitive attribute from frontend sensitive pointer types to backend `load/store` in-

structions generated for accessing them. Based on the access and pointer type, we select one from `ld_scp`, `ld_sdp`, `sd_scp`, and `sd_sdp`, and encode the remaining 15 TTag bits in the instruction's `funct3` and `imm` fields. Since these `imm` fields originally hold address offsets, we instrument `addi` instructions to handle non-zero offsets separately.

Object side, we default all memory to STag 0 and instrument `set_tag1/0` to set and clear CIDs for sensitive pointers at the start and end of their lifetime. Heap sensitive pointers live from their allocation to deallocation. When the allocated or deallocated region holds a sensitive pointer, the callsite casts between a sensitive pointer type (the region address) and a `void*` (the return value or argument of the allocation or deallocation function). We attach an intrinsic to such casts to carry the pointee types to the backend, then intercept this intrinsic in a function pass and instrument `set_tag1/0` instructions for sensitive pointers inside the region. Since the region address may be `NULL`, we do a `NULL` check before `set_tag1/0` at each instrumentation site. For array-shaped allocations, e.g., with `kcalloc()`, we repeat `set_tag1/0` for each element when the count is a compile-time constant, and otherwise emit a loop bounded by it. Stack sensitive pointers live from function start to end, so we instrument `set_tag1/0` at function entry and exit in a function pass. Global sensitive pointers live from module initialization to termination, so we instrument at global constructors and destructors in a module pass.

CID-aware bulk memory operation lowering. Sensitive pointers can be handled in bulk memory operations, e.g., via explicit `memset()`/`memcpy()` calls, or compiler-emitted aggregate initialization and copy. By default, they are lowered to original `load/store` that cannot access sensitive pointers, so we inspect the pointee type of their arguments and lower them to CID-aware versions that access sensitive pointers with `ld/sd_sc(d)p` instructions.

Linux transfer functions such as `copy_from_user()` move data across the kernel-user boundary as raw bytes, raising the same issue as `memcpy()` when the kernel object holds sensitive pointers. Because these functions are hard-coded assembly with certain instructions registered in the exception table, we cannot replace them with CID-aware versions. As a workaround, we leave them untouched, stage the data on the kernel stack at each callsite, and emit a CID-aware `memcpy()` to copy data between the stack buffer and kernel object.

6.3.3. Operating system support. SDP requires OS support for exception handling, CID clearing, context switching, page swapping, and a random allocator.

Exception handling. When a CID mismatch occurs, the hardware raises an exception to the OS. SDP has two modes: *enforcing* and *permissive*. In enforcing mode, the OS treats CID mismatches as attacks and terminates the offending process for user space violations, or panics for kernel space violations. In permissive mode, it instead logs the violation and continues the execution, which is useful for debugging and resolving false positives.

CID clearing. In program execution, we set and clear CIDs for sensitive pointers in pairs at the start and end of their lifetime. However, for cases like `__noreturn` functions that do not return to their callers and clear CIDs, e.g., `do_exit()`, there may be CIDs left uncleared, causing potential false positives. When a dead task is released, we walk the frame-pointer chain and clear CIDs for each frame.

Context switching. During context switches, the OS saves register states in memory, which may contain sensitive pointers. To mitigate their leakage or corruption, we protect the data structure storing these register states with SDP, e.g., `struct thread_struct` in Linux.

Page swapping. When a page is swapped out to disk, the OS collects its CIDs with `get_tag`, saves them in a data structure protected by SDP, and clears CIDs on the page with `clear_tag`. When the page is swapped back into a new page, the OS restores the CIDs with `set_tag1`.

Random allocator. We add a random allocator to allocate sensitive data of each type in a separate heap partition, an `sdp_cache`. On RISC-V Linux with the SV48 layout [35], we randomly choose the base address of each `sdp_cache` within the 32-TiB `vmalloc/ioremap` range, yielding sufficient entropy (45 bits) to defend against random guessing and brute-force attacks. We manage the object allocation within each `sdp_cache` with `gen_pool`.

For each object to allocate, we randomly decide (1) whether to add an adjacent tripwire block (25% chance), and (2) if so, whether to place it before or after the object (50% chance each). In tripwire blocks, we set every 8-byte slot as a sensitive pointer to act as a tripwire, using `set_tag1` with a TTag derived from the `sdp_cache` pointer, and clear the slots with `set_tag0` when deallocating the corresponding object. These random tripwires make random guessing or brute-force attacks even more impractical.

7. Security Analysis

In this section, we discuss SDP’s security guarantees and tradeoffs, analyze potential attacks, discuss if they can bypass or weaken SDP, and present case studies of recent Linux kernel exploitation techniques targeting sensitive data.

7.1. Security Guarantees

SDP does not prevent memory corruptions from being triggered, but contains the impact within a single compartment, either the default compartment or a sensitive one. Since sensitive pointers are only a small portion of all data in memory, vulnerabilities are probabilistically common in the default compartment but rare in sensitive ones. We empirically analyze 154 KernelCTF exploits [13] and find none of the exploited vulnerabilities falls in the sensitive compartments of our evaluated SDP kernels (Section 8.1), which is consistent with the probabilistic reasoning.

In the rare case when the vulnerability is in a sensitive compartment, SDP guarantees that attackers cannot access sensitive pointers in any other compartment, nor any data in

the default compartment. In the common case when it is in the default compartment, SDP guarantees that attackers cannot access sensitive pointers in any sensitive compartment, and accessing the hidden sensitive data is impractical.

For control-flow hijacking mitigation, SDP provides stronger security guarantees than in-production CFI mechanisms, such as Shadow Call Stack (SCS) [36] and KCFI [32] in the Linux kernel.

SCS could be bypassed if attackers with arbitrary read/write manage to locate and corrupt the shadow stack. SDP protects return addresses within a sensitive compartment inaccessible to attackers. None of the `ld/sd_scp` in this compartment is attacker-controllable, as they use `sp`-relative addressing with hardcoded offsets that attackers cannot modify.

KCFI restricts indirect callsites to invoke functions of expected types. When protecting function pointers, SDP checks CIDs when loading them for use and enforces that only pointers in the same compartment can be loaded, achieving the same mitigation effect as KCFI. Additionally, SDP requires exploiting a vulnerability in a sensitive compartment (the rare case) to corrupt a protected function pointer, while KCFI does not, so SDP provides stronger security guarantees than KCFI.

7.2. Tradeoff: Security, Manual Effort, Overhead

Overwriting indirect sensitive pointers is a valid attack path. We consider their protection in SDP policy, and our pointer type-based compartmentalization mechanism supports protecting them. However, similar to CPS and CPI [34], there is a tradeoff between the security benefit of protecting them and the manual effort and overhead.

As discussed in Section 7.1, protecting direct function pointers with SDP already provides stronger security guarantees than KCFI. Further protecting indirect function pointers only mitigates attacks that redirect them to another function pointer of the same type, which brings limited security benefit, but adds manual effort and overhead.

We therefore make indirect sensitive pointer protection a security hardening option in SDP, and developers can choose to enable it based on their security requirements and resource constraints.

7.3. Attack Analysis

We categorize possible attacks based on whether the vulnerability occurs in the default compartment (the common case) or a sensitive compartment (the rare case).

In the common case. *Memory scanning attacks.* Unable to leak sensitive pointers, attackers may instead scan memory to locate sensitive data, e.g., searching for the default value `"core"` to find `core_pattern`. However, SDP randomly allocates sensitive data across a large memory space. Sensitive pointers scattered throughout the space and random tripwires around sensitive data would trap attackers and force them to restart from scratch.

Information hiding attacks. Various attacks aim to locate hidden sensitive data in memory, through allocation oracles [37], thread spraying [38], crash-resistant memory probing [39], or side channels [40]. The first three attacks can be mitigated with proper system configuration and code practice, by limiting the maximum virtual memory and thread count per process, and enforcing non-crash-resistant exception handling. The last attack requires treating a sensitive pointer as a variable that affects control flow (e.g., a loop counter) to leak its value via a timing side channel. SDP mitigates it by disallowing normal `load` to access sensitive pointers, so they cannot be treated as other data like loop counters. Moreover, unlike prior work CPI [34] that stores metadata for locating sensitive data in the hidden region, SDP stores none, so even if attackers manage to locate a hidden region, they can hardly determine whether the target data resides inside or how to precisely locate it. The random tripwires around sensitive data further hinder precise localization.

Dependency attacks. SDP protects sensitive data and pointers, but not the dependent variables used to compute them, e.g., a pointer addend. SDP does not mitigate such attacks by default, as they are hard in practice: attackers must precisely locate these variables and catch the right timing to overwrite them. Still, developers who consider them a threat can protect them as sensitive data with SDP.

In the rare case. *CID collision attacks.* SDP allows sensitive pointer access instructions to only access pointers with matching TTags. Since TTags are calculated by hashing pointer types, collisions may happen, which attackers could exploit to access sensitive pointers of another type but with the same TTag. SDP reduces collisions by using a 16-bit wide TTag with a large hash space, and limits their impact by dedicating one TTag bit to separate code and data pointers, preventing an attacker-controlled `ld/sd_sdp` from accessing a code pointer and vice versa.

In both cases. *CID corruption attacks.* Object-side CIDs cannot be read or written through memory access instructions, so attackers cannot corrupt them with a `store`, regardless of where the vulnerability occurs. Only `set_tag1/0` and `clear_tag` can set CIDs, so corrupting CIDs requires either redirecting control flow to these instructions, which is mitigated by SDP or CFI mechanisms, or corrupting their address operands, which is mitigated by the security checks we add to `set_tag1/0` and by `clear_tag`'s limited usage. Attackers may also attempt Rowhammer attacks [41], [42] to flip STag bits and convert sensitive pointers to normal ones or the other way around. However, such attacks are improbable when control-flow hijacking is mitigated. They are also typically imprecise and may lead to random bit flips, which would trigger CID mismatch exceptions and stop the attack.

7.4. Case Study

We now present case studies of recent exploitation techniques aiming to escalate privileges in the Linux kernel,

from real-world exploits in Google KernelCTF [43] to recent research papers [44], [45].

Among the 154 KernelCTF exploits we analyzed [13], 105 redirect control flow to functions that set privilege-related fields. SDP can mitigate them by protecting return addresses and function pointers as sensitive pointers. The remaining 49 overwrite privilege-related fields or their pointers: 41 target `core_pattern`, 4 target `modprobe_path`, and 4 target `struct cred`. SDP can mitigate them by protecting the targeted data as sensitive data.

DirtyCred [44] introduces a new data-only privilege escalation attack. It exploits a heap vulnerability to free the attacker process's unprivileged `struct cred` and allocate a privileged one in the freed slot, making the kernel treat the attacker as privileged. SDP can mitigate this by protecting `struct cred`, then it becomes hidden, and attackers cannot locate and free it, which halts the attack.

RetSpill [45] relies on control-flow hijacking primitives to escalate privileges, which can be mitigated by protecting return addresses and function pointers with SDP. Additionally, it crafts malicious data in user space and exploits it once spilled to the kernel. SDP can also mitigate this by protecting the kernel data structures that preserve user space data, then the crafted data cannot be located and exploited.

8. Evaluation

We apply SDP to protect different program assets in Linux kernel v6.6 [46], and evaluate its security guarantees and performance impact on protecting sensitive data from data-only attacks and control-flow hijacking.

8.1. SDP Kernels

We build 6 SDP kernels with the following program assets protected:

- SDP-SD: 16 sensitive data fields and their pointers;
- SDP-RET: return addresses;
- SDP-FP: function pointers;
- SDP-FPIFP: direct and indirect function pointers;
- SDP-RETFP: return addresses and function pointers;
- SDP-SDRETFP: 16 sensitive data fields, return addresses, and function pointers.

Protection details. We protect the sensitive data and pointers in Table 2 with SDP and report the lines of code (LoC) of manual effort (Section 6.3.1) each requires.

Sensitive data. We protect 16 sensitive data fields that play critical security roles, are commonly attacked, cover various Linux components, and cannot be protected by existing mechanisms like post-init read-only memory [47]. Attackers corrupt these fields to achieve malicious goals such as privilege escalation, container escape, hardware resource manipulation (e.g., page permissions, memory mappings, and CPU register states), security mechanism bypass, audit log suppression, information leakage, and attack surface expansion.

TABLE 2. PROTECTED SENSITIVE DATA/POINTERS IN SDP KERNELS AND THE LOC OF MANUAL EFFORT EACH REQUIRES. # LOC CHANGES: [# SENSITIVE POINTER ANNOTATION + # ACCESS THROUGH POINTERS + # RANDOM ALLOCATION + # FALSE POSITIVE AVOIDANCE] # TOTAL.

Sensitive Data/Pointer	Component	Attack Goal	# LoC Changes
core_pattern	Core OS	Escalate privileges at core dump.	[1+5+13+0] 19
modprobe_path	Module	Escalate privileges at unknown-format file execution.	[2+3+14+0] 19
*(struct task_struct.cred)	Process	Escalate privileges at cred corruption.	[1+0+20+0] 21
*(struct task_struct.nsproxy)	Process	Escape namespace isolation at nsproxy corruption.	[1+0+9+0] 10
*(struct task_struct.fs)	Process	Escape chroot jails to get full filesystem access.	[1+0+11+0] 12
page tables	Core OS	Corrupt page table entries to change page permissions.	[5+1+305+0] 311
init_mm	Core OS	Redirect kernel page tables or corrupt memory mapping.	[2+44+20+0] 66
*(struct task_struct.mm)	Process	Redirect process page tables or corrupt memory mapping.	[1+0+7+0] 8
struct task_struct.thread	Process	Corrupt or leak register states of suspended threads.	[1+186+44+4] 235
sig_enforce	Module	Allow inserting unsigned kernel modules.	[1+24+6+0] 31
struct selinux_state	SELinux	Disable SELinux policy enforcement.	[2+10+8+0] 20
aa_g_audit	AppArmor	Disable audit logging to avoid leaving attack traces.	[1+5+8+0] 14
sysctl_unprivileged_bpf_disabled	BPF	Allow unprivileged users to load BPF programs.	[2+9+8+0] 19
sysctl_io_uring_disabled	io_uring	Allow new io_uring instance creation.	[1+15+8+0] 24
kptr_restrict	Core OS	Leak kernel pointers printed with %pK.	[2+12+8+0] 22
panic_on_oops	Core OS	Allow more attack attempts or leak info from oops logs.	[2+13+8+0] 23
return address	All	Hijack control flow at function return.	[0+0+0+0] 0
function pointer	All	Hijack control flow at function pointer invocation.	[0+0+0+94] 94

We follow Section 6.3.1 to protect these fields. The manual effort for false positive avoidance in protecting `struct task_struct.thread` (4 LoC) comes from accessing sensitive pointers in hard-coded assembly with `ld/sd_sdp` instructions. For per-process fields bound to `struct task_struct`, e.g., `*(struct task_struct.cred)`, we annotate their pointers in `task_struct` as sensitive, but do not propagate the sensitivity further to `task_struct` pointers, since the current process’s pointer resides in the `tp` register that attackers cannot corrupt. For fields used in early boot before `sdp_cache` exists, i.e., `page tables` and `init_mm`, we maintain boot-time versions for them and migrate them to `sdp_cache` once it is initialized.

Return addresses. The `gp` register is reserved for SCS in RISC-V Linux. We repurpose it to hold the return address location in function prologues and epilogues to optimize the instrumentation. We therefore modify the kernel to avoid leaking `gp` to user space or clobbering user space `gp` values.

Function pointers. To avoid false positives, e.g., due to function pointers as union members, we annotate the related function pointers with `__tag0` to force them insensitive.

Indirect function pointers. As discussed in Section 7.2, protecting them requires more manual effort, while offering limited security benefit. Since their protection relies on the same mechanism as the other sensitive pointers, for which we have demonstrated the required manual effort, we do not complete it for this kernel and skip its security evaluation. We still include it in the performance evaluation to show an estimated impact of the wide sensitivity propagation.

SDP kernel statistics. Table 3 shows the number of unique sensitive pointer types in SDP kernels, counted by walking all `load/store` in a pre-optimization pass. Protecting the 16 sensitive data fields adds 4 pointer types, for `sdp_cache` maintenance and pointer access of global or embedded heap sensitive data. Protecting function pointers

adds 4 more, as some bulk memory operations are lowered to CID-aware versions and expose more pointer types.

Table 3 also shows the static count of memory access instructions in each SDP kernel image. The “Other load/store” category covers all integer/floating-point load/store, load-reserved and store-conditional, their compressed counterparts, and atomic memory operations (counted as store). The ratio of `ld/sd_sc(d)p` among all load/store is generally low, indicating a low probability of exploitation (the rare case discussed in Section 7.3).

Table 4 shows the dynamic count of executed instructions during Linux startup for kernel and user space initialization. Kernel initialization starts at `start_kernel()` and ends after `do_sysctl_args()`. User space initialization starts after the successful execution of `run_init_process()` and ends when the login prompt is printed. We exclude SDP-FPIFP, as its manual effort is incomplete. All other SDP kernels execute with no false positives.

8.2. Security Evaluation

We build a functional emulator of SDP architecture on QEMU v9.1.1 [48] and evaluate SDP’s security with it.

Methodology. We reproduce exploits targeting the protected sensitive data and pointers to evaluate SDP’s effectiveness. Since public exploits are limited and not directly applicable to RISC-V, we build a character device that can arbitrarily read and write kernel memory through an `ioctl` (with original `load/store` or `ld/sd_sc(d)p` with a certain TTag) to mimic powerful real-world attacks.

These mimicked exploits cover all the attack paths discussed in Section 2, and both the common and rare cases discussed in Section 7.1. We show the exploitation steps, the manifestations on unprotected kernels, and on SDP kernels in Appendix Table 5. The device driver and mimicked exploits are released together with SDP prototype.

TABLE 3. NUMBER OF UNIQUE POINTER TYPES AND STATICALLY COUNTED INSTRUCTION NUMBERS IN SDP KERNELS. SENS. DPTR/SENS. CPTR: SENSITIVE DATA/CODE POINTER TYPES. RATIO: LD_SC(D)P AND SD_SC(D)P AMONG ALL LOAD AND STORE INSTRUCTIONS, RESPECTIVELY.

Image	# Pointer Types			# Static Instruction Counts									
	Sens. DPTr	Sens. CPTr	All	set_tag0	set_tag1	ld_scp	ld_sdp	sd_scp	sd_sdp	Oth. LD	Oth. ST	All	Ratio (%)
SDP-SD	11	0	6,029	32	28	0	1,422	0	48	494,363	316,354	2,397,275	0.29, 0.02
SDP-RET	0	1	6,025	28,913	28,976	28,912	0	28,976	0	465,496	286,874	2,508,291	5.85, 9.17
SDP-FP	0	1,777	6,029	28,591	28,239	8,174	0	2,853	0	488,765	319,647	2,542,320	1.64, 0.88
SDP-FPIFP	1,908	1,777	6,029	54,374	50,297	8,159	47,948	2,845	11,818	443,389	313,976	2,698,187	11.23, 4.46
SDP-RETFP	0	1,778	6,029	57,557	57,268	37,139	0	31,882	0	459,797	290,619	2,657,901	7.47, 9.89
SDP-SDRETFP	11	1,778	6,033	57,625	57,329	37,170	1,421	31,912	48	459,595	290,940	2,663,688	7.75, 9.90

TABLE 4. DYNAMICALLY COUNTED INSTRUCTION EXECUTION NUMBERS DURING LINUX STARTUP FOR KERNEL AND USER SPACE INITIALIZATION.

Image	Stage	# set_tag0	# set_tag1	# ld_scp	# ld_sdp	# sd_scp	# sd_sdp	# Oth. LD	# Oth. ST	# All
SDP-SD	Kernel	232	1,218	0	86,768	0	329	173,913,616	135,235,350	888,405,156
	User	29,816	31,757	0	2,057,801	0	6,963	1,285,780,886	781,477,596	6,906,234,889
SDP-RET	Kernel	2,131,048	2,131,185	2,131,027	0	2,131,185	0	171,964,985	132,896,033	896,370,007
	User	35,650,151	35,650,453	35,647,753	0	35,650,453	0	1,249,660,860	745,151,060	7,037,992,471
SDP-FP	Kernel	57,898	90,167	279,281	0	58,518	0	173,871,877	134,960,117	888,657,027
	User	610,125	616,123	2,309,735	0	621,089	0	1,280,502,071	777,874,917	6,888,857,621
SDP-RETFP	Kernel	2,192,231	2,224,637	2,413,656	0	2,192,988	0	171,679,072	132,817,950	896,936,623
	User	36,348,395	36,354,399	38,043,965	0	36,359,409	0	1,252,125,729	746,228,870	7,069,309,736
SDP-SDRETFP	Kernel	2,199,743	2,232,983	2,421,344	86,807	2,200,339	329	171,449,639	133,016,376	897,437,666
	User	36,341,959	36,350,596	38,007,003	2,039,071	36,323,469	6,963	1,248,170,944	744,777,404	7,057,566,110

Results. SDP effectively mitigates all the reproduced exploits. Attacks attempting to corrupt sensitive data directly are blocked when reading its pointers with mismatching CIDs. Those attempting to overwrite sensitive pointers are blocked when writing them with mismatching CIDs.

8.3. Performance Evaluation

We build a performance model that uses performance-equivalent instructions from the existing RISC-V ISA to model SDP’s new instructions. We compile SDP kernels based on this model and run them on a SiFive Unmatched [16] development board, with FU740 SoC and 16 GB RAM, running Ubuntu 25.04.

This performance model allows us to evaluate SDP performance on real hardware with full benchmark execution, and compare against the data-only attack mitigation baseline xMP [7] and the control-flow hijacking mitigation baselines SCS [36] (backward-edge CFI) and KCFI [32] (forward-edge CFI).

Methodology. As discussed in Section 6.2, we store object-side STags in spare ECC bits, and fetch STags and TTags alongside data with no extra memory access. CID checking runs in parallel with existing operations, adding no extra clock cycles. We therefore model ld/sd_sc(d)p with ld/sd instructions. Since ld/sd_sc(d)p carry TTag bits in their imm fields and cannot encode address offsets, the SDP compiler inserts an addi to compute the address before each such access with a non-zero offset (Section 6.3). We keep these addi instructions in the performance model and use a zero offset in the modeling ld/sd, so the model includes their overhead. For CID maintenance,

we model set_tag1/0 with sd zero, off(addr) instructions, as they share identical pipeline stages.

We use the same set of micro (LMbench-3.0 [17]) and macro (Phoronix v10.8.4 [18]) benchmarks as related work xMP [7]. For LMbench, we pin tasks to an isolated core and report the average of 5 runs to minimize variability. Phoronix measures whole-system performance, so we leave tasks on the default Linux scheduler without pinning them. It reruns each benchmark adaptively (3–20 runs) until the coefficient of variation falls below a threshold (1%) or the run limit is reached, and reports the average values.

Results. In Figures 4 to 9, bars show speedup or throughput normalized to the baseline, vanilla Linux v6.6, with error bars denoting ± 1 standard deviation across all runs (except for xMP’s reported results). We show geomeans across benchmarks next to the legends.

Protecting different program assets. In LMbench (Figure 4), SDP-SD, SDP-FP, and SDP-FPIFP show comparable performance relative to the baseline, averaging 0.98 $\times$, 1.00 $\times$, and 0.99 $\times$. SDP-RET shows a higher slowdown at 0.95 $\times$, due to frequent function invocations, most of which execute one set_tag1 and one set_tag0 to maintain return address CIDs. SDP-SDRETFP averages 0.94 $\times$, showing the performance of protecting the 16 sensitive data fields against both data-only attacks and control-flow hijacking.

The protection fault outliers are due to LMbench’s reporting logic: instead of reporting the raw protection fault latency it measures, it reports the raw latency minus a separately measured signal delivery cost to show the net cost of fault handling. However, the subtracted test takes a different execution path and is several times larger than the residual, so small noise or SDP-introduced differences in the two paths can cause large variations in the reported results.

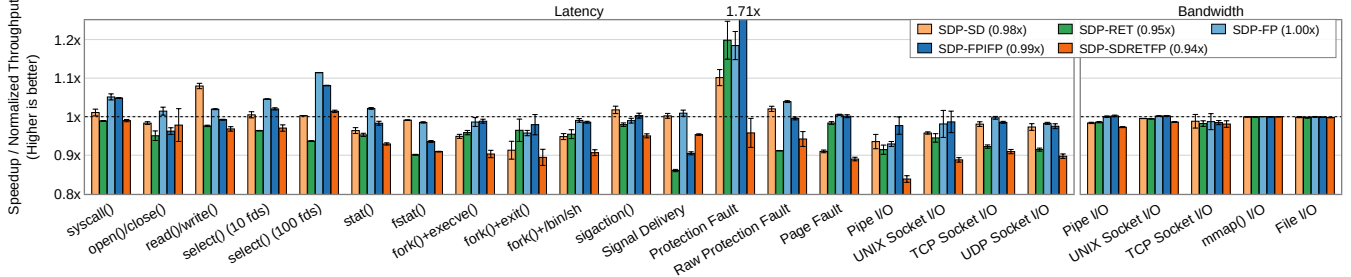


Figure 4. LMBench results of protecting different program assets with SDP.

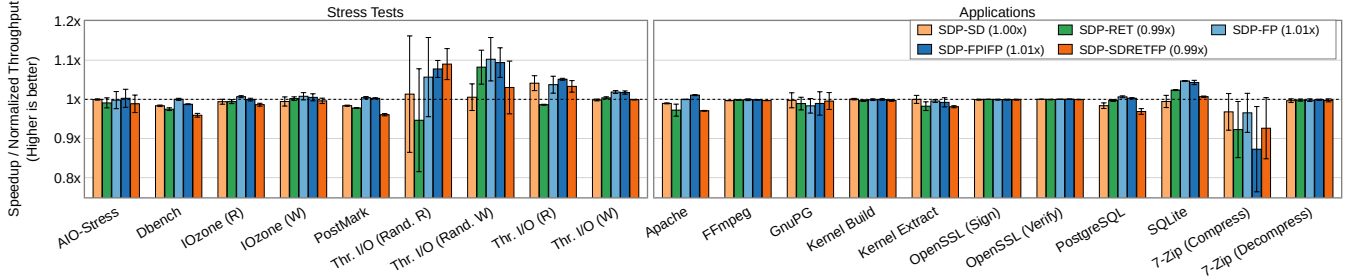


Figure 5. Phoronix results of protecting different program assets with SDP.

We therefore also measure and report the raw end-to-end protection fault latency without subtraction, which is more stable and reflects the actual executed path. We include the end-to-end latency instead of the subtracted one in geomean calculations, except in Figure 6, where xMP reports only the subtracted latency.

We also observe that SDP-FP, SDP-FPIFP, and SDP-SD speed up some benchmarks instead of slowing them down. We attribute this to the `sd` instructions we instrument to model `set_tag1/0`, which bring the corresponding cache lines into the L1 data cache and could improve performance in a manner analogous to software prefetching. It allows overlapping cache miss handling with the execution of independent instructions, and therefore reduces hardware stalls and improves overall hardware resource utilization.

In Phoronix (Figure 5), all SDP kernels perform comparably to the baseline, with SDP-SD, SDP-RET, SDP-FP, SDP-FPIFP, and SDP-SDRETFF averaging $1.00\times$, $0.99\times$, $1.01\times$, $1.01\times$, and $0.99\times$. This demonstrates that SDP efficiently protects different program assets in Linux with negligible performance impact on both stress tests and real-world user space applications.

SDP kernels show large deviations in three benchmarks: Threaded I/O Tester random read, random write, and 7-Zip Compression. The first two involve intensive random I/O, which is inherently unstable. The third runs LZMA’s match finder that roams a large dictionary with poor locality, so its working set far exceeds the cache. Run-to-run changes in physical page allocation shift page coloring and conflict miss counts, which causes variance in the results, especially on cores with small caches and low TLB associativity. Our evaluation core has a 32-KiB L1 data cache, a 40-entry L1

TLB, and a direct-mapped 512-entry L2 TLB [49], so it shows large deviations on both baseline and SDP kernels.

Comparison with xMP. We compare SDP against related work protecting sensitive data from data-only attacks, xMP [7]. xMP’s source release is incomplete, containing only partial primitives, and some primitives are inconsistent with the paper [50]. Therefore, we protect the same data xMP protects (`cred` and page tables, which constitute all the kernel data xMP evaluates), run the same benchmarks, and compare against its reported performance. Since xMP is evaluated on a different ISA, hardware platform, and Linux version, we normalize each system to its own vanilla baseline and regard the comparison as indicative.

Figures 6 and 7 show the comparison results. Protecting both `cred` and page tables, SDP averages $0.99\times$ of the baseline in LMBench and $1.00\times$ in Phoronix, whereas xMP averages $0.65\times$ and $0.94\times$, respectively. SDP thus outperforms xMP by 52% in LMBench and 7% in Phoronix.

Comparison with SCS and KCFI. We further compare SDP against existing backward- and forward-edge CFI mechanisms that can be applied to RISC-V Linux: software-based SCS [36] and KCFI [32]. RISC-V has ratified ISA extensions for CFI, but Linux currently supports them only for user space programs, not kernel protection [51], [52], and our evaluation board does not implement them, so we do not compare SDP against hardware-based CFI.

Figures 8 and 9 show the comparison results of SCS with SDP-RET, KCFI with SDP-FP, and their combinations in LMBench and Phoronix. SDP-RET and SCS add similar instrumentation to each function, so they perform comparably. SDP-FP marginally outperforms KCFI, and SDP-RETFF shows a small edge over SCS+KCFI.

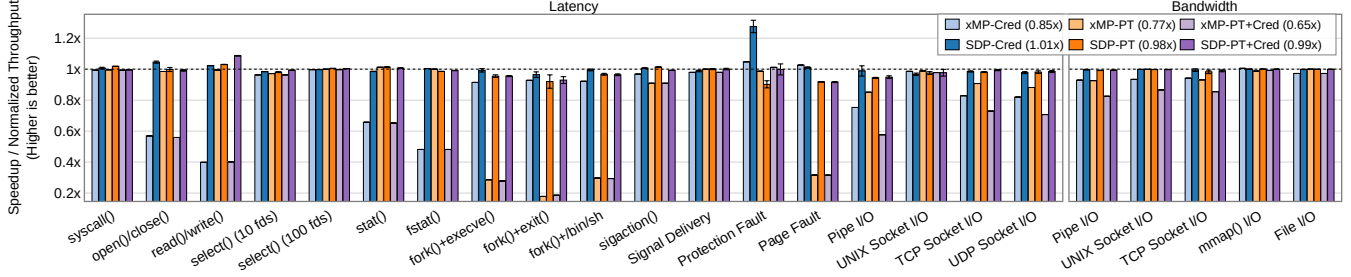


Figure 6. LMbench results of comparing SDP with xMP’s reported performance.

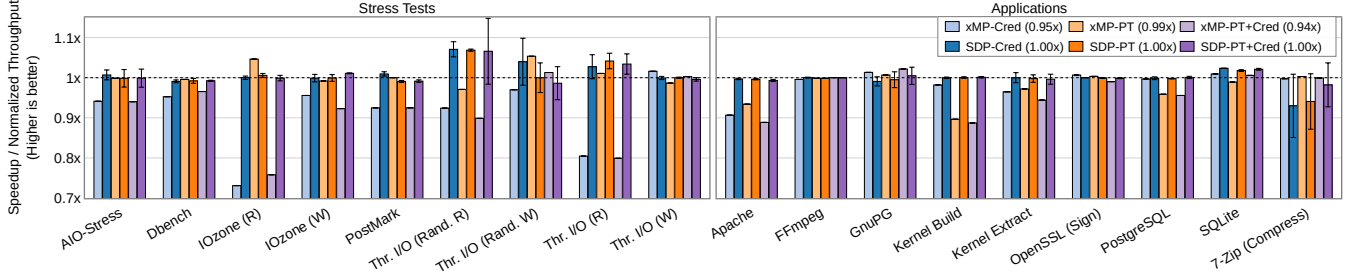


Figure 7. Phoronix results of comparing SDP with xMP’s reported performance.

9. Related Work

We categorize the related work into three groups according to their protection goals and summarize their differences in Appendix Table 6.

Sensitive data protection. xMP [7], PeTAL [8], and DOPE [53] share the same goal with SDP, but stack multiple mechanisms, compounding complexity, overhead, and security weaknesses. All three depend on CFI to mitigate control-flow hijacking. PeTAL protects sensitive data with ARM MTE, whose 4-bit tags are brute-forcible. xMP and PeTAL authenticate sensitive data pointers with HMACs and ARM PA, respectively, which are vulnerable to reuse attacks [12]. xMP and DOPE leave indirect sensitive data pointers unprotected. They also require manually instrumenting domain switches, which could widen access windows and violate the Least Privilege Principle. SDP provides a unified solution with strong security guarantees, negligible performance overhead, and limited manual effort (Appendix Table 6).

LKRG [54] detects rather than prevents sensitive data corruption, and misses any attack that corrupts and restores the data within a single checking interval.

Secure kernel allocation mechanisms like SeaK [55] mitigate heap layout manipulation, but attackers with arbitrary read and write capabilities can still locate and corrupt sensitive data.

Pointer protection. The following work protects only pointers and focuses primarily on write protection, while SDP protects both sensitive data and pointers, and provides both read and write protection for sensitive pointers.

CPI [34] isolates code pointers in a safe region accessible only via CPI intrinsics. SDP protects sensitive pointers in place, avoiding complex memory layout rearrangement.

ARM PA [9] authenticates pointers using authentication codes (PACs) stored in unused pointer bits. However, PAC can be modified by attackers and is vulnerable to reuse [12] and brute-force attacks. SDP also stores TTags in pointers, but ensures that they are inaccessible to attackers, as the corresponding STags are 1. STags are also inaccessible to attackers, so CID checking in SDP cannot be bypassed through attacker-controlled memory access instructions.

ZeRØ [25] protects pointers with a similar but different mechanism from SDP. Due to its metadata encoding, ZeRØ can distinguish at most 1,024 pointer types (65,536 in SDP), which is insufficient for complex programs like OS kernels (Table 3). Moreover, ZeRØ sets tags implicitly during pointer writes with weak checks and clears them at deallocation, leaving attackers the opportunity to unauthorizedly tag other locations by controlling a special `store`, or unauthorizedly clear tags by controlling the deallocation address or size. The pointer with a cleared tag can then be corrupted by a normal `store`. Although the special `load` that later reads the pointer fails the tag check, the corrupted value can still be used, since ZeRØ does not skip the faulty `load` and only raises an advisory exception, which does not terminate the program by default. In contrast, SDP sets object-side CIDs with dedicated instructions and performs CID checks before setting or clearing them, which mitigates unauthorized CID manipulation. It also enforces exceptions on `load` CID mismatches (in enforcing mode), further reducing the security risk of CID manipulation.

STAR [24] protects pointers by tagging data and instruc-

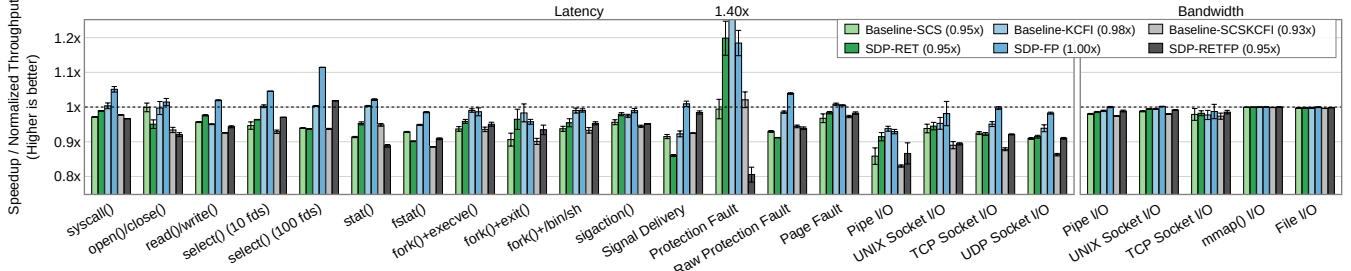


Figure 8. LMBench results of comparing SDP with SCS and KCFL.

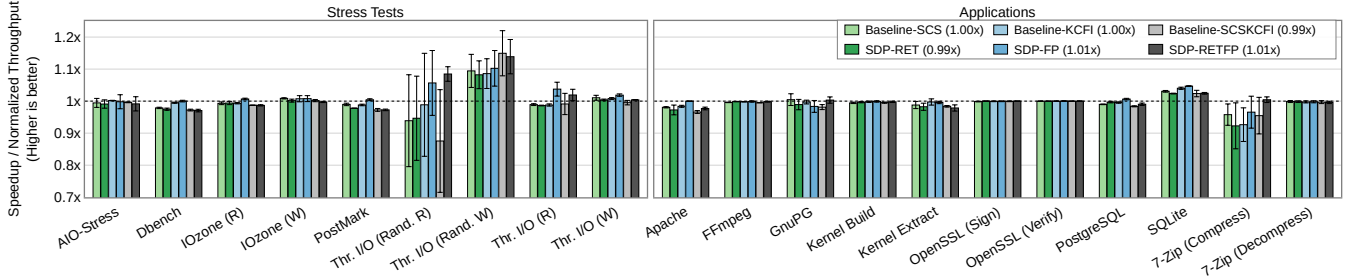


Figure 9. Phoronix results of comparing SDP with SCS and KCFL.

tions, but only coarsely, without distinguishing pointer types.

Memory safety violation mitigation. Mitigating memory safety violations aims to protect all data in memory, including sensitive data, so the following work has a superset of SDP’s goal. However, with the focus on all data, they may introduce significant overhead and manual effort that SDP mostly avoids, or provide weaker protection for sensitive data or pointers than SDP. SDP does not mitigate all violations, but applies strong protection on the small set of ultimate attack targets, efficiently and practically preventing attackers from achieving their malicious goals.

CHERI [4], [5] provides strong memory protection, but imposes substantial performance and memory overhead compared to SDP. SDP’s manual effort also stays far below CHERI’s (5,942 LoC for a still partially protected Linux kernel [56]), since CHERI must cover all pointers, whereas SDP covers only sensitive data and pointers.

Another category of related work locks the memory region a pointer can access and attaches a key to the pointer, so that only pointers with the correct key can access the region. ARM MTE [6] achieves this with memory tagging. It locks every 16 bytes in memory with a 4-bit tag and stores the matching key in pointers’ unused bits. However, the key is attacker-modifiable, and its small space can be easily brute-forced. C³ [22] encrypts part of the pointer, and uses it to generate the keystream for encrypting the pointed data. While this prevents pointer forgery, it only covers the heap. Unlike these approaches that bind pointers to data with memory tagging or encryption, SDP binds instructions to data (sensitive pointers) with matching CIDs determined at compile time, and ensures CIDs cannot be corrupted through attacker-controlled memory access instructions at runtime.

10. Conclusion

SDP introduces a unified solution to protect sensitive data against both data-only attacks and control-flow hijacking. We propose to randomly allocate sensitive data and enforce read and write protection on sensitive pointers through pointer type-based compartmentalization, which simplifies the overall system design and reduces overhead, while still providing strong security guarantees.

We prototype SDP extending the RISC-V ISA, evaluate its security guarantees by protecting 16 commonly attacked security-sensitive fields in the Linux kernel, and show its negligible performance impact with a set of micro (LMBench) and macro (Phoronix) benchmarks. The results demonstrate that SDP provides an effective and practical solution for sensitive data protection.

Our full prototype is open-sourced at <https://github.com/HexHive/SDP>.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments and constructive feedback. We are grateful to Chao Zhang, Shanqing Lin, Yuanlong Li, Tao Lyu, Trevor E. Carlson, Marius Momeu, and Mona Vij for the valuable discussions and suggestions. We also thank Miguel Peón-Quirós and David Atienza from EcoCloud and AVP-CP, respectively, for generously lending us the SiFive Unmatched development board, which enabled our performance evaluation on real RISC-V hardware. This work was funded, in part, by SNSF grant number 200021-236559 and a gift from Intel Corp.

References

- [1] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, pp. 48–62.
- [2] Anthropic. (2026) Anthropic’s coordinated vulnerability disclosure dashboard. [Online]. Available: <https://red.anthropic.com/2026/cvd/>
- [3] —. (2026) Measuring llms’ ability to develop exploits. [Online]. Available: <https://red.anthropic.com/2026/exploit-evals/>
- [4] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, “The CHERI capability model: Revisiting RISC in an age of risk,” in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 457–468.
- [5] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, “CHERI: A hybrid capability-system architecture for scalable software compartmentalization,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 20–37.
- [6] ARM. (2026) Memory tagging extension. [Online]. Available: <https://developer.arm.com/documentation/102925/0100/>
- [7] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, “xmp: Selective memory protection for kernel and user space,” in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 563–577.
- [8] J. Kim, J. Park, Y. Lee, C. Song, T. Kim, and B. Lee, “Petal: Ensuring access control integrity against data-only attacks on linux,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 2919–2933.
- [9] Qualcomm. (2026) Pointer authentication on armv8.3. [Online]. Available: <https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf>
- [10] Clang. (2026) Control flow integrity. [Online]. Available: <https://clang.llvm.org/docs/ControlFlowIntegrity.html>
- [11] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, “Fineib: Fine-grain control-flow enforcement with indirect branch tracking,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2023, Hong Kong, China, October 16-18, 2023*. ACM, 2023, pp. 527–546.
- [12] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J. Ekberg, and N. Asokan, “PAC it up: Towards pointer integrity using ARM pointer authentication,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 177–194.
- [13] Contributors. (2026) Kernelctf public exploits. Commit `eec2e575c80761eaeef089a842eaeef2841e2ebc7c`. [Online]. Available: <https://github.com/google/security-research/tree/master/pocs/linux/kernelctf>
- [14] RISC-V. (2026) The risc-v open-standard instruction set architecture. [Online]. Available: <https://riscv.org/specifications/ratified/>
- [15] QEMU. (2026) Qemu: A generic and open source machine emulator and virtualizer. [Online]. Available: <https://www.qemu.org/>
- [16] SiFive. (2026) Hifive unmatched. [Online]. Available: <https://www.sifive.com/boards/hifive-unmatched>
- [17] LMBench. (2026) Lmbench source code. Commit `11636a62695bb616b4454f8292ebfcef9eee57c2`. [Online]. Available: <https://github.com/intel/lmbench>
- [18] Phoronix. (2026) Phoronix test suite source code. Commit `f0365737ae42c2a4ae80c28491a8c0ac22a10f3c`. [Online]. Available: <https://github.com/phoronix-test-suite/phoronix-test-suite>
- [19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: Cold boot attacks on encryption keys,” in *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, P. C. van Oorschot, Ed. USENIX Association, 2008, pp. 45–60.
- [20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19.
- [21] H. Lefeuvre, N. Dautenhahn, D. Chisnall, and P. Olivier, “Sok: Software compartmentalization,” in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 3107–3126.
- [22] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney, “Cryptographic capability computing,” in *MICRO ’21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 253–267.
- [23] M. Ismail, A. Quach, C. Jelesnianski, Y. Jang, and C. Min, “Tightly seal your sensitive pointers with pactight,” in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 3717–3734.
- [24] R. T. Gollapudi, G. Yuksek, D. Demicco, M. Cole, G. Kothari, R. Kulkarni, X. Zhang, K. Ghose, A. Prakash, and Z. Umrigar, “Control flow and pointer integrity enforcement in a secure tagged architecture,” in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2974–2989.
- [25] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, “Zero: Zero-overhead resilient operation under pointer integrity attacks,” in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 999–1012.
- [26] Oracle. (2026) Hardware-assisted checking using silicon secured memory (ssm). [Online]. Available: https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html
- [27] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyklevich, and D. Vyukov, “Memory tagging and how it improves C/C++ memory safety,” *CoRR*, vol. abs/1802.09517, 2018.
- [28] H. Sakai, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, “Practical byte-granular memory blacklisting using califorms,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 558–571.
- [29] M. Unterguggenberger, D. Schrammel, L. Lamster, P. Nasahl, and S. Mangard, “Cryptographically enforced memory safety,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 889–903.
- [30] R. H. Gumpertz, “Combining tags with error codes,” in *Proceedings of the 10th Annual Symposium on Computer Architecture, 1983*, H. W. L. Jr., T. Agerwala, H. H. Heilborn, H. Aiso, L. Thorelli, J. Baer, and M. Tokoro, Eds. ACM, 1983, pp. 160–165.
- [31] M. B. Sullivan, M. T. I. Ziad, A. Jaleel, and S. W. Keckler, “Implicit memory tagging: No-overhead memory safety using alias-free tagged ECC,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, Y. Solihin and M. A. Heinrich, Eds. ACM, 2023, pp. 67:1–67:13.

- [32] Linux. (2023) Merge patch series "riscv: Kcfi support". Commit 7f7d3ea6eb00bd329a6f2fe3f1c7596c4e783e1. [Online]. Available: <https://github.com/torvalds/linux>
- [33] LLVM. (2026) Llvm source code. Commit 7ba7d8e2f7b6445b60679da826210cdde29eaf8b. [Online]. Available: <https://github.com/llvm/llvm-project>
- [34] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 147–163.
- [35] Linux. (2026) Virtual memory layout on risc-v linux. [Online]. Available: <https://www.kernel.org/doc/html/v6.6/riscv/vm-layout.html#risc-v-linux-kernel-sv48>
- [36] —. (2023) Merge patch series "riscv: Scs support". Commit 24005d184aaa80984e0511c4ec6e6a0860fdddb8. [Online]. Available: <https://github.com/torvalds/linux>
- [37] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 121–138.
- [38] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining information hiding (and what to do about it)," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds. USENIX Association, 2016, pp. 105–119.
- [39] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.
- [40] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. E. Shrobe, S. Sidiroglou-Douskos, M. C. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 781–796.
- [41] Y. Kim, R. Daly, J. S. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 361–372.
- [42] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting correcting codes: On the effectiveness of ECC memory against rowhammer attacks," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 55–71.
- [43] Google. (2026) Kernelctf vulnerability reward program. [Online]. Available: <https://google.github.io/security-research/kernelctf/rules.html>
- [44] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 1963–1976.
- [45] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupe, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 3093–3107.
- [46] Linux. (2026) Linux source code. Commit ffc253263a1375a65fa6c9f62a893e9767fbebfa. [Online]. Available: <https://github.com/torvalds/linux>
- [47] —. (2026) Post-init read-only memory. Commit c74ba8b3480da6ddaea17df2263ec09b869ac496. [Online]. Available: <https://github.com/torvalds/linux>
- [48] QEMU. (2026) Qemu source code. Commit 0ff5ab6f57a2427a3e83969b2e7dd71e04caae39. [Online]. Available: <https://github.com/qemu/qemu>
- [49] SiFive. (2026) Sifive fu740-c000 manual. [Online]. Available: <https://www.sifive.com/document-file/freedom-u740-c000-manual>
- [50] virtsec. (2020) xmp: Selective memory protection for kernel and user space. [Online]. Available: <https://github.com/virtsec/xmp#disclaimer>
- [51] Linux. (2026) Shadow stack to protect function returns on risc-v linux. [Online]. Available: <https://docs.kernel.org/arch/riscv/zicfiss.html>
- [52] —. (2026) Tracking indirect control transfers on risc-v linux. [Online]. Available: <https://docs.kernel.org/arch/riscv/zicfilp.html>
- [53] L. Maar, M. Schwarzl, F. Rauscher, D. Gruss, and S. Mangard, "DOPE: domain protection enforcement with PKS," in *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. ACM, 2023, pp. 662–676.
- [54] LKRG. (2026) Linux kernel runtime guard. [Online]. Available: <https://lkrg.org/>
- [55] Z. Wang, Y. Guang, Y. Chen, Z. Lin, M. V. Le, D. K. Le, D. Williams, X. Xing, Z. Gu, and H. Jamjoom, "Seak: Rethinking the design of a secure allocator for OS kernel," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024.
- [56] K. Wang, D. Kasatkin, V. Ahlrichs, L. Auer, K. Hohentanner, J. Horsch, and J. Ekberg, "Cherifying linux: A practical view on using CHERI," in *Proceedings of the 17th European Workshop on Systems Security, EuroSec 2024, Athens, Greece, 22 April 2024*. ACM, 2024, pp. 15–21.

Appendix A. Reproduced Exploit Details

TABLE 5. EXPLOITATION STEPS, MANIFESTATIONS ON UNPROTECTED KERNELS, AND ON SDP KERNELS.

Target	Steps	Unprotected Kernel	SDP Kernel
core_pattern	Overwrite it with a malicious script as the pipe handler, then trigger a core dump via a deliberate segfault in a child process.	The malicious script runs as root, with uid: 0, gid: 0, and whoami: root.	Exception at core_pattern pointer load.
modprobe_path	Overwrite it to point to a malicious script, then trigger modprobe by executing a file with an unknown magic number.	The malicious script runs as root, with uid: 0, gid: 0, and whoami: root.	Exception at modprobe_path pointer load.
*(struct task_struct.cred)	Follow it to the cred struct and zero out all uid/gid fields.	The process uid becomes 0. /bin/sh spawns a root shell.	Exception at .cred load.
*(struct task_struct.nsproxy)	Run in isolated namespaces via unshare, overwrite its pointer, and redirect it to init_nsproxy. Observe mnt/net/uts inode numbers and hostname before/after.	The process switches from the isolated namespaces to the host's init namespaces.	Exception at .nsproxy store.
*(struct task_struct.fs)	chdir into a marker dir inside a chroot, overwrite its pointer, and redirect it to init_fs. Observe getcwd() and readlink(/proc/self/{cwd,root}) before/after.	The process escapes the chroot jail into the host's init filesystem (root /).	Exception at .fs store.
page tables	Follow init_mm.pgd to walk the kernel page table (pgd→pud→pmd→pte) and locate the leaf entry that maps a kernel code page. Change its permission bits from read-execute to read-write-execute.	The kernel code page becomes writable after a TLB flush. Writing a nop to it succeeds.	Exception at init_mm.pgd pointer load.
init_mm	Follow its .pgd field to walk the kernel page table and locate the leaf entry that maps a kernel code page. Change its permission bits from read-execute to read-write-execute.	The kernel code page becomes writable after a TLB flush. Writing a nop to it succeeds.	Exception at init_mm pointer load.
*(struct task_struct.mm)	Follow it to the mm_struct and overwrite its .start_brk (heap start address) to corrupt the process's memory mapping.	The corrupted .start_brk shows in /proc/self/stat.	Exception at .mm load.
struct task_struct.thread	Follow it to the thread_struct and overwrite its .ra field with the address of a dmesg logging gadget for a sleeping thread, to redirect its control flow on waking.	The sleeping thread's execution is redirected to the gadget after waking, logging to dmesg.	Exception at .thread pointer load.
sig_enforce	Overwrite it from 1 to 0, to disable kernel module signature enforcement. Load an unsigned module before/after.	The unsigned module gets rejected before, but loads after.	Exception at sig_enforce pointer load.
struct selinux_state	Overwrite its .enforcing field from 1 to 0, to disable SELinux policy enforcement. Run sestatus before/after.	sestatus reports SELinux is permissive (no enforcement).	Exception at selinux_state pointer load.
aa_g_audit	Overwrite it from 0 to 2, to silence audit logs and hide policy violations. Check the module parameter before/after.	The module parameter reports AppArmor in quiet audit mode.	Exception at aa_g_audit pointer load.
sysctl_unprivileged_bpf_disabled	Overwrite it from 2 to 0, to allow unprivileged users to load BPF programs and enlarge the attack surface. Query it via sysctl before/after.	The value reads 0, allowing all processes to load BPF programs.	Exception at the sysctl_unprivileged_bpf_disabled pointer load.
sysctl_io_uring_disabled	Overwrite it from 1 to 0, to allow new io_uring instance creation and enlarge the attack surface. Query it via sysctl before/after.	The value reads 0, allowing all processes to create new io_uring instances.	Exception at the sysctl_io_uring_disabled pointer load.
kptr_restrict	Overwrite it from 1 to 0, to disable the redaction of kernel pointers printed via %pK. Check dmesg before and after.	Kernel pointers via %pK leak as raw addresses in dmesg.	Exception at kptr_restrict pointer load.
panic_on_oops	Overwrite it from 1 to 0, to allow more attack attempts or leak information from oops logs. Deliberately trigger a kernel segfault that causes an oops, and observe the kernel behavior.	The kernel logs the oops but keeps running, not panicking.	Exception at panic_on_oops pointer load.
return address	Overwrite a return address to point to a commit_creds (&init_cred) gadget, to corrupt task_struct.cred and escalate to root privileges at function return.	The gadget executes at return. The process uid becomes 0. /bin/sh spawns a root shell.	Exception at return address store.
function pointer	Overwrite a direct function pointer softirq_action. action in softirq_vec[TIMER_SOFTIRQ] to a dmesg logging gadget, to redirect control flow at timer interrupts.	The gadget executes on every timer interrupt and prints logs to dmesg.	Exception at .action store.
	Overwrite an indirect function pointer struct file.f_op to point to a fake file_operations, whose .read is the address of a commit_creds (&init_cred) gadget, to escalate to root privileges at pipe read().	The gadget executes at pipe read(). The process uid becomes 0. /bin/sh spawns a root shell.	Exception at .read load.

Appendix B.

Comparison with Related Work

TABLE 6. COMPARISON OF SDP AGAINST RELATED WORK. SCOPE: SENSITIVE DATA (SD), DIRECT/INDIRECT SENSITIVE DATA POINTERS (DPTR), DIRECT/INDIRECT CODE POINTERS (CPTR); ✓, ○, ✗ FOR FULLY, PARTIALLY, NOT COVERED.

Related Work	Mechanism	Scope			Security	Overhead		Main Manual Effort
		SD	DPtr	CPtr		Main Operations ^a	Main Metadata	
Sensitive data protection								
xMP [7]	SD: isolate with hypervisor-enforced page permissions. DPtr: authenticate with HMACs.	✓	○	✗	Attacker-modifiable HMACs are vulnerable to reuse attacks. Indirect DPtrs are not signed. Manual instrumentation for SD and DPtrs could widen access windows or miss protection. Relies on CFI.	Per SD access window: 2 domain switches. Per DPtr access: 1 HMAC. CFI.	Per domain: 1 EPT view carrying full permissions for all domains, plus 1 key frame.	Specify and allocate SD. Manually instrument domain entry/exit and DPtr sign/auth.
PeTAL [8]	SD: tag with ARM MTE. DPtr: authenticate with ARM PA.	✓	✓	✗	Attacker-modifiable 4-bit MTE tags and PACs are vulnerable to brute-force and reuse attacks. Provides match-all tags that bypass checks. Relies on CFI.	Per SD access: 1 tag check. Per DPtr access: 1 PAC sign/auth. CFI.	MTE tags: 4 bits per 16 bytes. 16-byte memory alignment.	Specify entry tables and identification rules. Verify the identified scope.
DOPE [53]	SD: isolate with Intel PKS. DPtr: validate on load against the owner address stored in the SD.	✓	○	✗	Indirect DPtrs are unvalidated. Stacks are unprotected. Access windows are widened for performance. Only 3 coarse domains; all SD with the same protection (e.g., write-protected) shares one. Relies on CFI.	Per SD access window: 2 PKS domain switches. Per DPtr access: 1 owner validation. CFI.	Per SD object: 1 added owner field, 2 dummy pages for one variant, or a shadow object for another variant.	Specify and allocate SD. Adjust object layout and add owner fields. Manually instrument domain entry/exit based on reported access sites.
LKRG [54]	SD: compare against a saved reference (a hash or a full copy), periodically and on access.	○	✗	✗	Detects but does not prevent SD corruption. Misses corrupt-and-restore attacks within one check interval. Bypassable by design.	Per interval (15s): check all SD. Per intercepted read: 1 check. Per legal write: 1 update.	Per SD object or a group of objects: 1 saved reference, a hash or a full copy.	None, provided as a Linux kernel module.
SeaK [55]	SD: allocate in a random region with guard pages, reusable only by the same allocation site.	○	✗	✗	Mitigates heap layout manipulation only. Accesses to SD are not checked. Attackers can still leak DPtrs and corrupt SD.	Per allocation/free site: 1 eBPF handler with 2 map accesses.	Per SD object: ≥1 exclusive page and a shared guard page (up to 8 KB).	Specify SD objects to protect.
Pointer protection								
CPI [34]	Isolate CPtrs and their metadata in a safe region accessible only via CPI intrinsics.	✗	✗	✓	Its x86-64 implementation hides the safe region rather than isolating it, so it can be located and corrupted [40]. The prototype enforces spatial memory safety only, not temporal safety.	Per CPtr access: 1 metadata access. Per CPtr dereference: 1 bound check.	Per CPtr: 4 words, while the unoptimized implementation reserves 4 words for every 8 bytes in the address space.	Handle custom allocators, assembly, and JIT code.
PARTS [12]	Authenticate with ARM PA, using pointer types as modifiers.	✗	✓	✓	Attacker-modifiable PACs are vulnerable to reuse attacks. All pointers of one type share a modifier.	Per pointer access: 1 PAC sign/auth.	None, PACs occupy unused pointer bits.	Skip authentication for some pointers to work with legacy code.
ZeRØ [25]	Tag each 8 bytes. For pointers, encode types in tags and allow only dedicated instructions to access them.	✗	✓	✓	Distinguishes 1,024 pointer types. Allows unauthorized metadata setting and clearing (Section 9). Allows a permit-list to bypass checks.	Per load/store: 1 tag check (in parallel with existing operations).	L1 cache line: 2 bytes per 64 bytes. L2/L3 cache line and memory: 1 bit per 64 bytes.	Specify permit-list entries. Handle third-party libraries.
STAR [24]	Tag each 4 bytes with a coarse type and each instruction with the type it or its operands must have. Disallow mismatch.	✗	✓	✓	Distinguishes only 4 types (data, data pointer, code pointer, and return address). In-type corruptions are not detected.	Per instruction: 1 tag decode. Per load/store: 1 data tag cache access and check.	Memory: 2 bits per 4 bytes. Instruction: 1 tag-carrying nop per 3 instructions. A dedicated data tag cache and TLB.	Handle assembly and JIT code.
Memory safety violation mitigation								
CHERI [4], [5], [56]	Equip every pointer with a tag-guarded unforgeable capability, and check its bounds and permissions on every dereference.	✓	✓	✓	Enforces unforgeable pointers and spatial memory safety. Temporal safety requires additional revocation sweeps.	Per load/store: 1 bounds and permission check. Per pointer access: 1 capability access. Per pointer arithmetic or bounds setting: 1 capability update.	Per pointer: 1 or 3 extra words (128-bit or 256-bit capability). Memory: 1 tag bit per 16 or 32 bytes (in ECC bits or an in-memory tag table). 16-byte memory alignment.	Handle every C idiom that capabilities break (integer casts, raw copies, overreads, pointer-size assumptions) and every site the compiler cannot certify (assembly, allocators, global-data relocation) [56].
ARM MTE [6], [27]	Lock every 16 bytes with a 4-bit tag; the pointer carries the matching key. Disallow mismatch on every access.	✓	✓	✓	Only 16 tag values, so an unauthorized access succeeds with probability 1/16. Attacker-modifiable in-pointer tags can be easily brute-forced. The Linux kernel covers only heap objects; stacks and globals are untagged.	Per load/store: 1 tag check. Per allocation and deallocation: 1 tag set.	Memory: 4 tag bits per 16 bytes. 16-byte memory alignment.	Make custom allocators tag-aware. Untag pointers wherever the tag bits break functionality (e.g., comparison, range check). Handle interfaces that reject tags (e.g., DMA).
C ³ [22]	Encrypt each pointer returned per allocation. Encrypt the data it points to with that pointer, so the data decrypts to garbled values with corrupted pointers/data.	○	○	○	Garbles rather than traps: a violation corrupts data instead of reliably raising an exception. Pointers can be reused, which gives plaintext access to the data. Heap only; stacks and globals are unencrypted.	Per allocation: 1 pointer encryption. Per load/store: 1 pointer decryption and 1 data decryption/encryption.	2 new registers for per-process keys. Per load buffer entry: 24 bits. Per store buffer entry: 25 bits.	Make custom allocators return encrypted pointers. Handle overreads. Pass plaintext data across unencrypted boundaries (kernel, I/O, shared memory).
SDP	SD: randomly allocate, guarded with random tripwires. DPtr/CPtr: compartmentalize based on pointer types, accessible only by dedicated instructions.	✓	✓	✓	For SD, 45-bit random allocation entropy with tripwires, mitigating random guessing and brute-force attacks. For DPtr/CPtr, distinguishes 65,536 pointer types and mitigates CID corruption. Provides stronger security than in-production CFI (SCS and KCFI).	Per load/store: 1 CID check (in parallel with existing operations). Per sensitive pointer lifetime: 1 CID set, 1 CID clear.	Memory: 1 bit per 8 bytes (in spare ECC bits). Cache: 1 byte per 64-byte cache line. Store queue and store buffer: 1 bit per entry.	Specify and allocate SD. Annotate direct DPtrs. Handle uninstrumentable sites (unions, assembly, non-standard C type usage), only for sensitive pointers, instead of every pointer/data.

^a We list main operations affecting performance instead of reported overhead, which is incomparable across platforms, benchmarks, and scopes.

Appendix C. Ethics Considerations

None.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper addresses the problem of protecting sensitive data in program memory against both data-only and control-flow attacks. The authors develop a unified approach to mitigate both attacks by both hiding the data (via random allocations) and protecting access to sensitive pointers. The authors present their design of SDP, which offloads pointer compartmentalization to hardware while involving the compiler and OS to manage the compartmentalization and randomized data allocation. The authors implement their prototype for RISC-V and LLVM, and demonstrate its low performance overhead when applied to protect sensitive data in the Linux kernel.

D.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

- 1) The paper provides a valuable step forward in an established field. While prior works have provided solutions to protect sensitive data, they incur higher performance overhead and require combining their data-only attack protections with separate CFI solutions. The authors propose a new approach to jointly address both attacks by hiding sensitive data and protecting sensitive pointers. The authors develop a software-hardware co-design to realize this approach, achieving low performance overhead while imposing limited burden on developers.

D.4. Noteworthy Concerns

The evaluation does not directly compare SDP-SDRETFP as a unified solution against any combined solutions, such as xMP used together with a CFI solution.

Appendix E. Response to the Meta-Review

We considered comparing SDP against the combination of xMP, SCS, and KCFI. However, this combination is not evaluated in the xMP paper, and we cannot evaluate it ourselves, since xMP's source release is incomplete, providing only partial primitives, and some primitives are inconsistent with the paper.

We compared SDP against each mechanism of the combination separately: xMP for data-only attacks, and SCS and KCFI for control-flow hijacking. SDP incurs substantially lower overhead than xMP and comparable overhead to SCS and KCFI. Since the overhead of the combined solution would compound, we expect SDP to outperform it.