ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks

Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, and Simha Sethumadhavan
Department of Computer Science
Columbia University
New York, NY, USA
{mtarek, miguel, evgeny, simha}@cs.columbia.edu

***Abstract*—A large class of today's systems require high levels of availability and security. Unfortunately, state-of-the-art security solutions tend to induce crashes and raise exceptions when under attack, trading off availability for security. In this work, we propose ZeRØ, a pointer integrity mechanism that can continue program execution even when under attack. ZeRØ proposes unique memory instructions and a novel metadata encoding scheme to protect code and data pointers. The combination of instructions and metadata allows ZeRØ to avoid explicitly tagging every word in memory, eliminating performance overheads. Moreover, ZeRØ is a deterministic security primitive that requires minor microarchitectural changes. We show that ZeRØ is better than commercially available state-of-the-art hardware primitives, e.g., ARM's Pointer Authentication (PAC), by a significant margin. ZeRØ incurs zero performance overheads on the SPEC CPU2017 benchmarks, and our VLSI measurements show low power and area overheads.**

***Index Terms*—Exploit Mitigation, Pointer Integrity, Memory Safety, Code-Reuse Defenses, Caches.**

I. Introduction

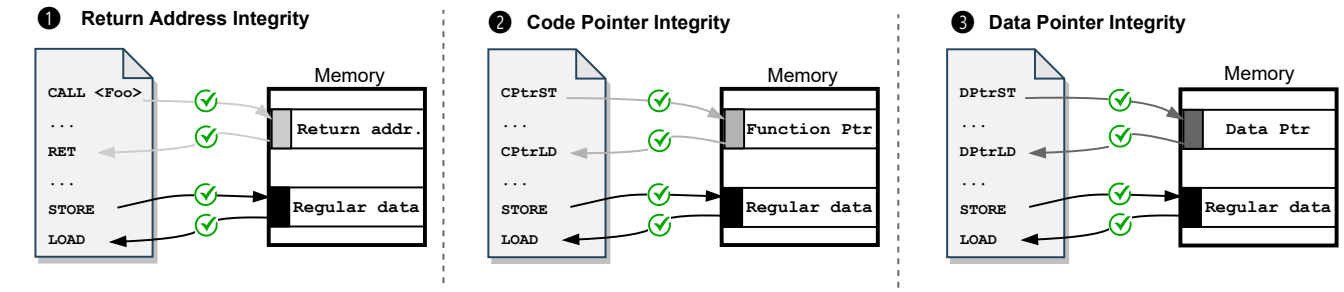
Pointers give programmers the raw ability to work with particular memory locations. The power and flexibility of pointers makes programs written in C and C++ very efficient as long as programmers are careful with their usage. Unfortunately, errors in pointer usage (e.g., out-of-bounds access) can lead to memory corruption vulnerabilities [47]. These memory corruption vulnerabilities have provided attackers with significant opportunities for exploitation. For example, attackers abuse memory safety vulnerabilities to overwrite code pointers and hijack the control flow of the program [41], [6], [16], [5]. Similarly, attackers target data pointers to build up sequences of operations (aka data-oriented gadgets) without modifying the program's control flow [20]. The prevalence of pointer manipulation attacks against modern software has prompted processor manufacturers to implement hardware mitigation primitives, such as Intel's CET [21] and ARM's PAC [37]. For example, PAC uses cryptographic message authentication codes (MACs) to protect the integrity of pointers, namely return addresses, code pointers, and data pointers. Unfortunately, PAC's usage of cryptographic primitives presents a non-zero performance and energy penalty. In addition, PAC remains vulnerable to speculative execution attacks where arbitrary pointers can be speculatively authenticated [17].

In this paper, we present ZeRØ, a hardware primitive that preserves pointer integrity at no additional performance cost. In traditional processors, memory instructions can freely access any memory location. There are no restrictions on the type of operands used by a memory instruction. We observe that this behavior is fundamental for attackers to craft their exploits. As a result, ZeRØ introduces unique sets of memory instructions for the different categories of pointers that make up a program (i.e., code and data). Having specific memory instructions for code pointers, data pointers, and regular data allows ZeRØ to enforce access control rules that maintain pointer integrity when under attack.

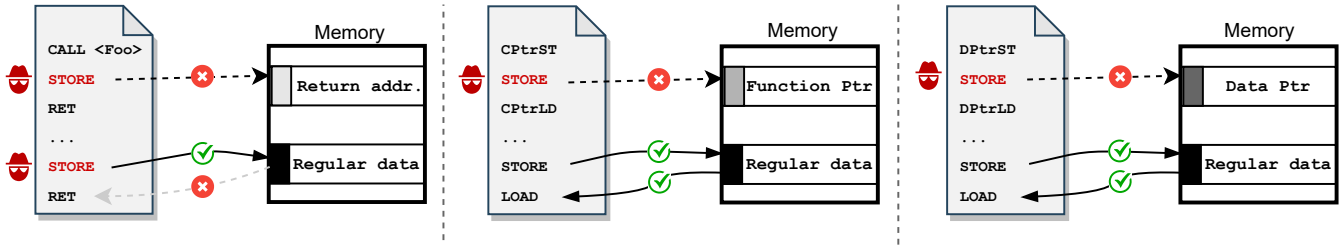
To better understand ZeRØ's security guarantees, let us consider the example in Figure 1. Under normal program execution, ZeRØ enforces three different classes of data integrity as shown in Figure 1(a), namely return address integrity, code pointer integrity, and data pointer integrity. Return address integrity aims at preventing the attackers from overwriting return addresses on the stack (aka return oriented programming, or ROP [41], [6]). Return address integrity is provided by extending the functionality of regular `CALL` and `RET` instructions to mark return addresses in memory and prevent other memory instructions from accessing them. As shown in Figure 1(b) - ❶, an attacker can attempt to overwrite (i.e., `STORE`) the return address and hijack the control flow of the program. As return addresses are marked such that they can only be accessed by `CALL/RET` pairs, ZeRØ prevents an attacker from hijacking control-flow.

ZeRØ provides code and data pointer integrity by introducing new pairs of memory instructions that are only allowed to access code and data pointers, respectively. For example, we use `CPtrST/CPtrLD` instructions for exclusively accessing code pointers as shown in Figure 1(a) - ❷. If an attacker attempts to overwrite a code pointer using a regular memory instruction (e.g., `STORE`) as shown in Figure 1(b) - ❷, ZeRØ prevents the memory access from occurring. ZeRØ maintains data pointer integrity in the same way as code pointers by introducing specific `DPtrST/DPtrLD` instruction variants as shown in Figure 1(a) - ❸.

Unlike prior work, which tags every word in memory to identify different program assets (e.g., code and data pointers) [46], [14], ZeRØ uses a novel metadata encoding scheme



(a) ZeRØ enforces access control rules that maintain pointer integrity for return addresses, function pointers, and data pointers.



(b) ZeRØ mitigates code-reuse attacks through its use of access control preventing regular `STORE`s from corrupting pointers.

Fig. 1: A high level overview of how ZeRØ's pointer integrity mechanism works.

that allows it to precisely store all the required metadata to identify different program assets with just a single bit per every cache line in L2 and main memory (less than 0.2% memory overheads).

ZeRØ additionally offers resilient operation under pointer integrity attacks in the following way: if an attacker attempts to overwrite a pointer using a regular memory instruction, ZeRØ rejects the violating memory access and continues program execution. If there is a need for more forensics, ZeRØ shares the address and operands of the violating instruction with the operating system by using an advisory exception. Unlike traditional exceptions, our advisory exceptions do not crash the running program unless the program is configured to do so. This way we prevent the attacker from abusing our defense to launch a denial-of-service attack.

We implement ZeRØ's software changes using the LLVM compiler infrastructure to emit our new memory instructions depending on pointer types with no false positives. Our experimental results on the SPEC CPU2017 benchmark suite indicate that the software overheads of ZeRØ are 0% compared to a baseline. Additionally, our VLSI implementation results show that ZeRØ can be efficiently added to modern processors with negligible performance, area, and power overheads. Unlike other pointer authentication solutions, ZeRØ does not need to dedicate an energy budget to cryptographic co-processors [31], [37], [30] or standalone shadow stacks [21].

II. Background

In this section, we provide an overview of memory corruption attacks and define our threat model.

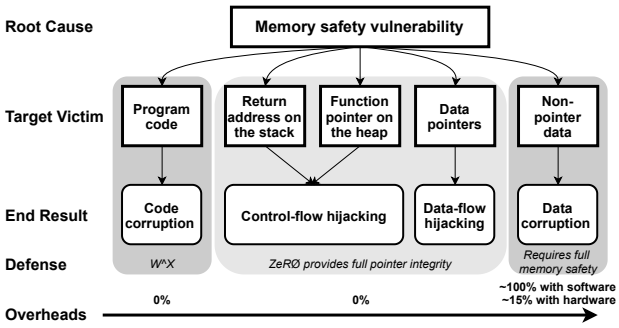


Fig. 2: Memory corruption targets, end results, and typical defenses. ZeRØ prevents the most common attack vectors by providing full pointer integrity with no performance overheads.

A. Memory Corruption Attacks

Figure 2 shows a taxonomy of memory corruption attacks in memory unsafe languages such as C and C++. The root cause of all memory corruption attacks is memory safety vulnerabilities such as buffer overflows and use-after-frees [47]. Once attackers have access to such vulnerabilities, they can target different program assets to achieve various end goals.

Code Corruption Attacks. Traditional approaches for exploiting memory vulnerabilities aimed at either (i) overwriting program instructions in memory with an attacker's payload or (ii) dumping the attacker's code discretely to the program stack and executing it. Nowadays, code corruption attacks are ineffective due to the widespread deployment of W^X [12]. In other words, an attacker cannot overwrite program data (i.e., code is marked as readable and executable but not writable)

and cannot write and execute their own code (i.e., data is marked as readable and writable but not executable).

Control-Flow Hijacking Attacks. This line of attacks (aka code reuse attacks or CRAs) exploits memory vulnerabilities to overwrite code pointers stored in memory. Corrupting a code pointer can cause a control-flow transfer to anywhere in executable memory. Code pointers include return addresses on the stack and function pointers anywhere in memory. As code pointers are stored in program memory (stack and heap), they are a common target for attackers. For example, return oriented programming (ROP) [41], [6] corrupts return addresses, whereas call- and jump-oriented programming [16], [5] corrupt function pointers (or indirect code addresses in general). To mount a CRA, the attacker has to first analyze the code to identify the attack gadgets, or sequences of instructions in the victim program that end with a return or jump instruction. Second, the attacker uses a memory corruption vulnerability to inject a sequence of target addresses corresponding to a sequence of gadgets. When the function returns (or a code pointer is dereferenced), it moves to the location of the first gadget. As that gadget terminates with a control flow instruction (e.g., return), it transfers program execution to the next gadget, and so on. As CRAs execute existing instructions belonging to the program, they are not prevented by W^X.

Data-Flow Hijacking Attacks. In contrast to control-flow hijacking attacks, data-oriented programming (DOP) attacks can cause malicious end results without changing the control flow of the program. Prior works show that manipulating data pointers in memory is sufficient for the attacker to achieve arbitrary computations on program input [20], [23], [36]. As DOP attacks do not alter the program control flow, they can easily bypass all control-flow integrity solutions. Thus, DOP is an appealing attack technique for future run-time exploitation defenses.

Data Corruption Attacks. This last class of attacks targets non-pointer data items while stored in memory. Examples include manipulating program flags to bypass selective checks and changing configuration parameters [9]. Mitigating non-pointer data corruption attacks requires full memory safety solutions, which come with high performance overheads.

B. Threat Model

Adversarial Capabilities. We assume that the adversary is aware of the applied defenses and has access to the source code, or binary image, of the target program. Furthermore, the target program suffers from memory safety-related vulnerabilities that allow the adversary to read from, and write to, arbitrary memory addresses. The attacker's objective is to (ab)use memory corruption and disclosure bugs, mount a code-reuse attack, and achieve privilege escalation. Furthermore, we include DOP [20] attacks in our threat model. We exclude pure data corruption attacks from our threat model as they target non-pointer data. This limitation applies to prior work as well [31], [7], [30], [14]. Due to their prominence, we include speculative execution attacks in our threat model [27].

Hardening Assumptions. We assume that the underlying operating system (OS) is trusted. If the OS is compromised and the attacker has kernel privileges, the attacker can execute malicious code without making CRAs; a simple mapping of the data page as executable will suffice. However, our technique can be applied to the operating system code itself for protecting code and data pointers. We assume that ASLR and W^X protection are enabled—i.e., no code injection is allowed (non-executable data), and all code sections are non-writable (immutable code). Thus, attacks that modify program code at runtime, such as rowhammer [26] and CLKSCREW [48], are out of scope.

Secrets. Unlike prior work, ZeRØ requires no secret parameters or configuration keys. The security is purely derived from runtime enforcement.

III. The ZeRØ System Overview

In this section, we describe how ZeRØ enforces pointer integrity for different program assets: return addresses, code pointers, and data pointers.

Return Address Integrity. In order to prevent return-oriented programming attacks, ZeRØ protects return addresses on the stack by extending the functionality of regular `CALL` and `RET` instructions to mark return addresses in memory and prevent program loads and stores from accessing them. When a `CALL` instruction is executed, the return address is pushed to the stack alongside the function arguments. ZeRØ sets 2 bits of metadata in the L1 data cache to 01 to mark the 8B return address as *protected*. When a `RET` instruction is executed, the return address is moved from the stack to the program counter if and only if it has the metadata bits set to 01. Once the metadata bits are verified, program execution moves to the new address and ZeRØ sets the metadata bits in the L1 data cache to 00 to mark the memory location as a regular location (i.e., *non-protected*). If any other `LOAD` or `STORE` instruction tries to access a memory location while its metadata bits are set to 01, the hardware generates an advisory exception, effectively preventing return addresses from being leaked or overwritten. The advisory exception is used to notify the system administrator of the access violation without crashing the running process.

Function Pointer Integrity. Similar to return addresses, ZeRØ uses metadata in the L1 data cache to mark function pointers. In order to accurately identify memory instructions that are supposed to access function pointers, ZeRØ uses compiler support and proposes two special instructions, Code Pointer Load (`CPtrLD`) and Code Pointer Store (`CPtrST`), to access function pointers. `CPtrST` marks the function pointer location as *protected* on the first use and assigns a unique state, 10, to it to distinguish function pointers from return addresses. Only `CPtrLD` instructions are allowed to load function pointers from those protected locations. ZeRØ generates an advisory exception if any regular memory instruction is

used to access a memory location that has its metadata bits set to 10.

Data Pointer Integrity. Data pointers work analogously to function pointers. Similarly, ZeRØ proposes two special instructions, Data Pointer Load (`DPtrLD`) and Data Pointer Store (`DPtrST`), to access data pointers. The functionality of these two instructions mirrors the usage of the code pointer variant as described above. While stored in memory, data pointers are assigned a unique L1 metadata state, 11, to avoid confusing them with other protected items (i.e., return addresses and function pointers). We elaborate more on the layout of our ZeRØ metadata and how it is propagated to main memory in Section V.

Pointer-Flow Integrity. In addition to distinguishing between different program assets (i.e., code pointers, data pointers, and regular data), ZeRØ achieves finer protection granularity by distinguishing between elements of the same program asset. To do so, ZeRØ encodes the pointer type in the spare bits (10 bits in our current prototype) of the pointer while executing `DPtrST`. We then verify that the pointer type matches the expected type at a `DPtrLD` location. The pointer type is assigned at compile time and does not require points-to analysis. Two pointers are compatible if their type is the same. As the types are encoded at `DPtrST`/`DPtrLD` sites, an attacker cannot use a vulnerable `DPtrST` instruction to corrupt data pointers of incompatible types, thus reducing the attack surface. The same approach is also applied to code pointers to prevent the attackers from confusing incompatible function pointers. In this case, the function type is used as a unique type per `CPtrST`/`CPtrLD` site.

In the next three sections, we describe the required instruction set extensions, hardware changes, and compiler support needed for ZeRØ.

IV. Instruction Set Extensions

One key aspect in ZeRØ's design is the ability to isolate code and data pointers in memory such that they are not corrupted by attacker-controlled memory instructions. Thus, ZeRØ extends the instruction set architecture to operate exclusively with code and data pointers.

• **`CPtrST/CPtrLD <R1>, <R2>`**: These instructions stand for Code Pointer Store and Code Pointer Load, respectively. Similar to traditional stores and loads, `CPtrST`/`CPtrLD` use two register operands. The values in registers `R1` and `R2` point to the store/load address and source/destination register as usual. These instructions are emitted by the compiler only to store/load code pointers. The compiler encodes the code pointer type in the upper bits of `R2`. Upon executing this instruction, the hardware sets/checks the corresponding metadata bits in the L1 data cache and matches the pointer type against the one stored in the upper bits of the memory location (i.e., the store/load address).

• **`DPtrST/DPtrLD <R1>, <R2>`**: These instructions stand for Data Pointer Store and Data Pointer Load, respectively. Similar to `CPtrST`/`CPtrLD`, they are emitted by the compiler to store/load data pointers. Upon executing this instruction, the hardware sets/checks the corresponding metadata bits in the L1 data cache and verifies the pointer type, as described above.

TABLE I: Actions taken on various instructions based on the memory location state. 00 represents regular data, 01 represents return address, 10 represents code pointer (i.e., specifically function pointers), and 11 represents a data pointer. X represents "Don't Care".

Instruction	Metadata State	Action
`CALL`	00	Set the metadata to 01.
	01	Invalid. Cannot overwrite a return address.
`RET`	00	Invalid. Cannot return from a non-taken address.
	01	Set the metadata to 01.
	10	Invalid. Cannot return from a function pointer.
	11	Invalid. Cannot return from a data pointer.
`CPtrST`	00	Set the metadata to 10.
	01	Invalid. Cannot overwrite a return address.
	11	Invalid. Cannot overwrite a data pointer.
`CPtrLD`	10	Load code pointer.
	00	Invalid. Cannot load a non-code pointer.
	X1	Invalid. Cannot load a non-code pointer.
`DPtrST`	00	Set the metadata to 11.
	01	Invalid. Cannot overwrite a return address.
	10	Invalid. Cannot overwrite a code pointer.
`DPtrLD`	11	Load data pointer.
	10	Invalid. Cannot load a non-data pointer.
	0X	Invalid. Cannot load a non-data pointer.
`LOAD/STORE`	00	Load/Store a non-pointer data item.
	01	Invalid. Cannot access a return address.
	10	Invalid. Cannot access a code pointer.
	11	Invalid. Cannot access a data pointer.
`ClearMeta`	01	Invalid. Cannot free stack memory.
	11	Set the metadata to 00.
	10	Set the metadata to 00.

• **`ClearMeta <R1>, <R2>`**: Code and data pointers corresponding metadata bits should be cleared when memory is freed. To support this functionality, we add a Clear Pointer Metadata (`ClearMeta`) instruction that takes two register operands. The value in register `R1` points to the starting address of a 64B cache line. The value in register `R2` is a mask to the corresponding 64B cache line, where 1 allows and 0 disallows changing the state of the corresponding byte. The mask is used to perform partial updates of metadata within a cache line. This instruction is treated similarly to a `STORE` instruction in the processor pipeline since it modifies the architectural state of data bytes in a cache line. Upon executing a `ClearMeta` instruction, the metadata of the target cache line in the L1 data cache is cleared.

Additionally, ZeRØ extends the implementation of regular `CALL` and `RET` instructions to set and check the validity of return address metadata state in the L1 data cache. This functionality is necessary to guarantee the integrity of return addresses. Unlike the other cases discussed prior, this feature does not require a special instruction or additional compiler support. There is no need to explicitly clear the return address metadata as they are cleared upon executing the `RET` instruction. Table I summarizes the actions taken on various instructions based on the memory location state.

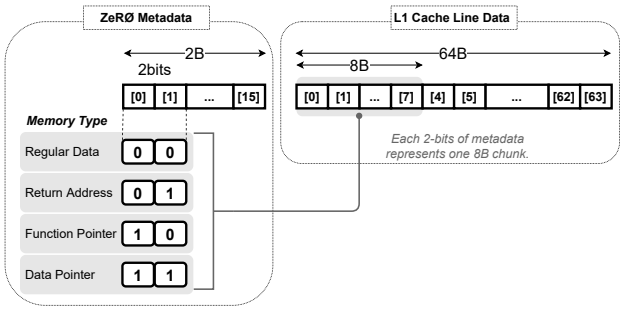


Fig. 3: ZeRØ's metadata encoding in the L1 data cache. ZeRØ uses a 16-bit vector to indicate whether a chunk of eight bytes is a return address, function pointer, data pointer, or regular (non-pointer) data.

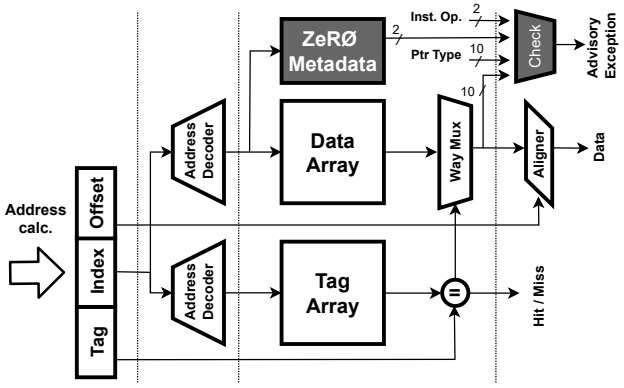


Fig. 4: Pipeline diagram for the L1 cache hit operation. The shaded blocks correspond to ZeRØ components.

V. Microarchitecture Design

In this section, we describe the microarchitectural changes that are required to enable ZeRØ.

L1 Data Cache Modifications. Figure 3 shows our L1 data cache metadata encoding. ZeRØ uses a 16-bit vector to identify the locations of return addresses, function pointers, and data pointers in a cache line. For example, each two bits of the metadata bit vector represent the state corresponding to each aligned 8B of the cache line. An 8B chunk can either be a return address (01), function pointer (10), data pointer (11), or regular data (00). Our bit vector introduces a 2B storage overhead per 64B cache line (a 3.125% storage for the L1 data cache). As shown in Figure 4, if a load/store accesses a protected byte (which is determined by reading the corresponding bit vector), an advisory exception is recorded to be processed when the load/store is ready to be committed.

Exception Handling Circuitry. For certain program functions or libraries, it might be desirable to suppress exceptions (e.g., when the program intentionally accesses pointers with regular `LOAD`/`STORE` instructions). ZeRØ provides hardware support for suppressing the advisory exceptions by using a permit-list. When a binary is loaded the OS writes the starting address and size of the permitted functions/libraries to the permit-list in the exception handling circuitry. Then, when a ZeRØ exception occurs, the hardware checks if the PC of the current memory access instruction is covered by the permit-list or not. If the PC is permitted, the advisory exception is suppressed. Otherwise, the advisory exception is raised and the faulting PC and memory address are passed to the OS exception handler so that they can be used for reporting or investigation purposes. In our design, we use an 8-entry permit-list, where the entry size is 12 bytes (eight bytes for the function starting address and four bytes for the size).

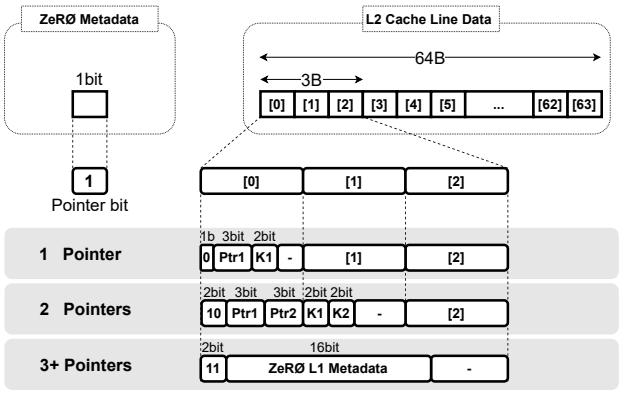


Fig. 5: ZeRØ's metadata encoding in L2/L3 data cache and main memory. Ptr1/Ptr2 encodes the offset of the pointer in the cache line, whereas K1/K2 encodes its type (return address, code pointer, or data pointer). ZeRØ uses a single bit of metadata to identify protected cache lines.

L2/L3 Cache Modifications. Figure 5 shows a schematic view of our L2/L3 data cache and main memory metadata encoding. ZeRØ uses a compressed format that requires 1 bit of metadata per 64B cache line (0.2% storage overheads) in L2/L3 caches and main memory. The key idea is that if a cache line has a protected memory location (i.e., return address, code pointer, or data pointer), it will have at least two unused bytes (i.e., the upper 16-bits of the pointer). We use 6 bits from the pointer's upper bits to encode its metadata. For example, if one pointer appears in the cache line, we use 3 bits to store its offset within the line and 2 bits to define its type (return address, code pointer or data pointer). We set the first bit to zero to easily identify this case. The original contents of the first 6 bits of the cache line are moved to the upper 6 bits of the pointer. Finding the location of this pointer requires scanning the bit vector for the occurrence of its state (e.g., 11 for data pointers). This operation is implemented with a priority encoder. We repeat the same approach if two assets of any type exist in a particular cache line.

A natural question to ask is: How do we handle the case in which multiple types of pointers exist in a cache line? For example, a cache line might have three or more code/data pointers. In this case, we have more unused upper pointer bits than needed. Thus, we use 2 bits for recognizing the case and 16 bits for storing the traditional ZeRØ L1 metadata for the entire line. To distinguish formatted lines from regular ones,

```
1: Read the bit vectors of the evicted line and OR them
2: if result is 0 then
3:     Evict the line as is and set its Pointer bit to zero
4: else
5:     Set the Pointer bit to one
6:     Get the location of the first 3 protected addresses
7:     Store the first 6, 12, or 18 bits in
        the locations obtained in 6
8:     Fill the first 6, 12, or 18 bits based on Figure 5
9: end
```

Algorithm 1: L1-to-L2 cache line transformation.

we use our single metadata bit (aka the `Pointer` bit) per cache line as an indicator. If the `Pointer` bit is set to one, that means we have pointers in the cache line. Otherwise, the cache line is normal (i.e., requires no processing).[1]

For DRAM, we store the additional bit per cache line into spare ECC bits, similarly to prior work [35], [40], [38]. We note that the DDR5 standard DIMMs use 80-bit channels, which provides ample space for additional metadata. For non-ECC DRAMs, ZeRØ's eight bytes per 4KB page can be stored in a disjoint location in memory at no additional cost.

L1 to/from L2 Transformation Module. ZeRØ uses two different formats: one for the L1 data cache and another for the L2/L3 data caches. As a result, a transformation module is needed to switch between the two formats while cache lines are moving between L1 and L2 in both directions. While the L1-to-L2 transformation module is not on the critical path (only invoked when cache lines are evicted from L1 to L2), the L2-to-L1 transformation module is on the critical path of the processor load operation. Thus, the transformation needs to be carefully designed in order to avoid adding latency to the L2 data cache access.

Algorithm 1 shows the high-level process of the L1-to-L2 transformation module. Figure 6 shows the block diagram of the same module. The process starts by ORing all bits from the input L1 bit vector to detect whether a protected address (return address, code pointer, or data pointer) exists in the cache line or not. If the result equals zero (i.e., no protected addresses are detected), we simply set the `Pointer` bit (L2 ZeRØ metadata in Figure 6) to zero. If any protected address is detected, we fill in the cache line header according to Figure 5. Priority encoders are used to find the index of the first three protected addresses, if they exist. We use the aforementioned locations to store the original contents of the first 6, 12, or 18 bits of the cache line (Line 7 in Algorithm 1) using a cross bar and combinational logic.

Algorithm 2 and Figure 7 show the high-level process and block diagram of our L2-to-L1 transformation module. If the cache line has its `Pointer` bit set to one, we check the least significant 2 bits of the first byte to identify the encoding case and reconstruct the original contents of the first 6, 12, or 12 bits of the cache line accordingly. We evaluate the latency and area overheads of our transformation modules in Section VIII-A.

Load/Store Queue Modifications. Since the `CALL`/`RET` instructions generate a store/load micro-op as part of their regular functionality on CISC systems, there is a chance of a load to store forwarding between the stored return address from the `CALL` instruction to a subsequent in-flight load instruction, violating our return address integrity. To avoid this scenario, ZeRØ extends load/store queue entries with 12 bits that specify whether the entry is associated with a return address, function pointer, data pointer, or regular data (2 bits) in addition to its pointer type (10 bits). This way entries marked as return addresses (or code/data pointers) are written as part of a `CALL` (or `CPtrST`/`DPtrST`) instruction and can only be forwarded to loads that are part of a `RET` (or `CPtrLD`/`DPtrLD`) instruction, respectively. To provide tamper-resistance against side-channel attacks, ZeRØ forwards the value zero from those entries to any matching in-flight load instructions and marks them as potential violators. An advisory exception is thrown only when the potential violating instructions are committed to avoid any false positives due to misspeculation. The checking operation is performed in parallel to the regular address matching process with no performance impact.

VI. Software Design

In this section we describe the memory management, compiler, and operating system support necessary to enable ZeRØ.

A. Memory Management

ZeRØ is agnostic to the memory allocator. ZeRØ intercepts any program calls to `free()`/`delete[]` and emits `ClearMeta` instructions to clear code- and data-pointers metadata from the free'd regions if it exists. Additionally, ZeRØ emits `ClearMeta` instructions on function returns to cleanup the stack frame.

B. Compiler Support

Pointer Integrity. To provide pointer integrity, we need to accurately identify `LOAD` and `STORE` instructions that access pointer values. To do so, our current prototype uses the Clang/LLVM compiler infrastructure to replace program code- and data-pointer loads and stores with our new instructions, `CPtrLD`/`CPtrST` and `DPtrLD`/`DPtrST`.

In order to protect code pointers that are initialized prior to runtime (e.g., entries in C++ virtual tables), we add a `handleGlobals` function that emits `CPtrST` instructions for all global pointers and invoke it at the start of the `main` function as part of program initialization. This way we protect all code pointers that have no explicit `STORE` instructions executed at runtime.

Pointer-Flow Integrity. To prevent pointer confusion between data pointers, we encode the type of the pointer in its most significant 10 bits prior to executing `DPtrLD`/`DPtrST`. We use the pointer's LLVM ElementType, which depends on the type of the pointed-to data structure. The ElementType

[1] We note that ZeRØ's metadata bits can be completely hidden on systems that use caches with error-correction-codes (ECC) support. The techniques from Gumpertz [18] can be used to store ZeRØ's metadata bits for free without compromising the typical ECC functionality. We leave this extension to future work.

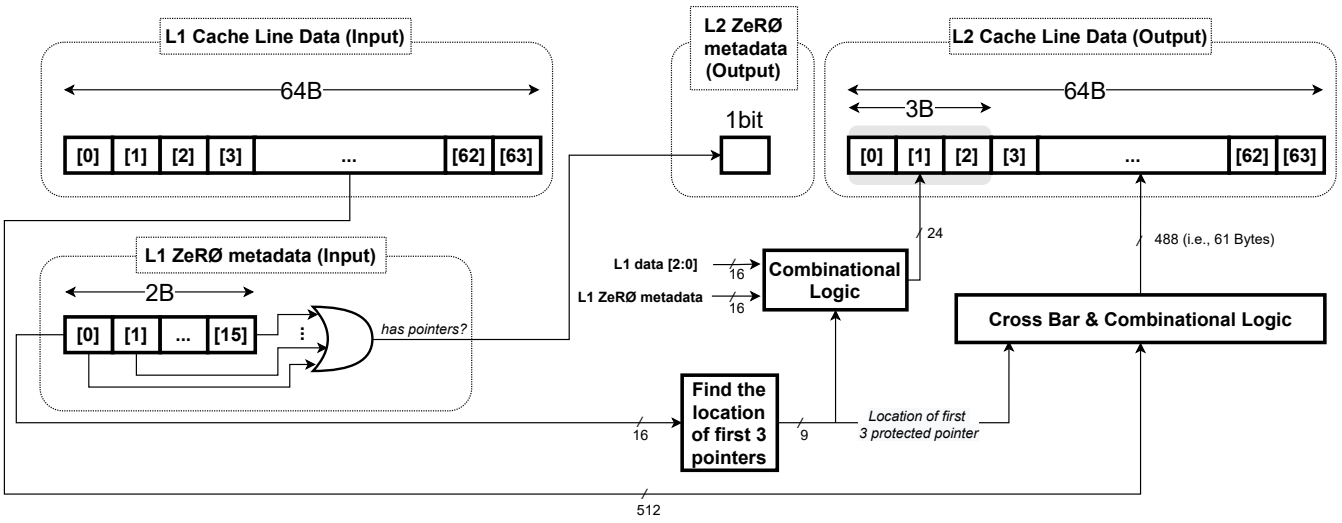


Fig. 6: Block diagram of the L1-to-L2 transformation module that is used during the spill operation. The left hand side shows the input L1 cache line data and the corresponding L1 ZeRØ bit vector.

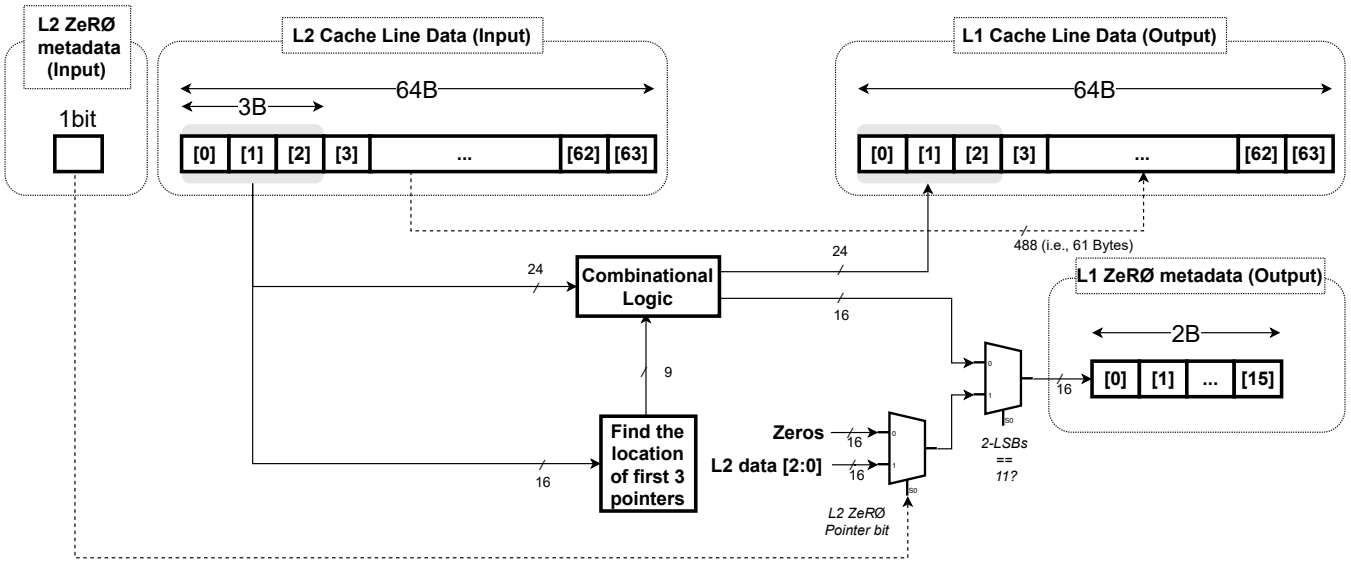


Fig. 7: Block diagram of the L2-to-L1 transformation module that is used during the fill operation. The left hand side shows the input L2 cache line data and the corresponding single `Pointer` bit of ZeRØ.

```
 1: Read the Pointer bit for the inserted line
 2: if result is 0 then
 3:     Set the entire bit vector to [0]
 4: else
 5:     Check the least significant 2 bits of byte 0
 6:     if result is 11 then
 7:       Copy bit[2-18] to the L1 bit vector
 8:       Get the location of the first 3 protected addresses
 9:       Set the data of bit[2-18] to the upper
           6 bits of location obtained in 8
10:     else
11:       Set the metadata of addresses[Ptr[1-2]] to K[1-2]
12:       Set the data of the first 12 bits to the most
            significant 6 bits of byte[Ptr[1-2]]]
13:     end
14: end
```

Algorithm 2: L2-to-L1 cache line transformation.

of each pointer is readily available and does not require points-to analysis. Similarly, we encode code pointer types in the most significant 10 bits of a code pointer prior to executing `CPtrLD`/`CPtrST` to prevent the attackers from confusing incompatible function pointers. In this case, we use the function type as a unique code pointer type.

Return Address Integrity. No compiler support is needed for return address integrity as ZeRØ extends the functionality of traditional `CALL`/`RET` instructions.

Finally, a recent work [43] shows that it is feasible to track data pointers in hardware with no compiler support. This pointer tracking feature should enable ZeRØ to relax its compiler support requirement for data pointer integrity.

C. Operating System Support

Advisory Exceptions. When ZeRØ's hardware detects an access violation, it throws an exception once the instruction becomes non-speculative. Our exceptions are advisory in nature. In other words, they do not halt program execution. Instead, they just notify the operating system of the invalid behavior and continue program execution (after rejecting the violating memory access).[2] ZeRØ provides hardware support for suppressing the advisory exceptions by using a permit-list, as described in Section V. For example, it might be desirable to add functions that copy plain bytes (e.g., `memcpy` and `memmove`) to the permit-list as they may generate exceptions upon accessing protected addresses (e.g., pointers). Wherever possible, our compiler pass emits type-aware copying functions that do not need any special exception handling. Of the 16 SPEC CPU2017 C/C++ benchmarks, only two (`502.gcc_r` and `526.blender_r`) have cases where a permit-list is needed. For the rest of the benchmarks, our compiler pass successfully identifies the copied operands types and emits our special instructions for copying the protected fields and regular memory access instructions for the non-pointer fields.

Page Swaps. ZeRØ requires 1 bit of metadata per 64B cache lines. When a page is swapped out from main memory, the page fault handler needs to store the metadata for the entire page into a reserved address space managed by the OS; the metadata is reclaimed upon swap in. The kernel has enough address space in practice (the kernel's virtual address space is 128TB for 64-bit Linux with 48-bit virtual address space) to store the metadata for all the processes on the system since the size of the metadata is minimal (8B for a 4KB page or 0.2%).

Stack Unwinding. The C standard includes the `setjmp` and `longjmp` programming interface, which can be used to add exception-like functionality to C. `setjmp` saves the current environment including the return address and stack pointer to a memory buffer (`jmp_buf`) while `longjmp` restores the previously saved environment from `jmp_buf`. To guarantee return address (and stack pointer) integrity while saved in `jmp_buf`, ZeRØ instruments `setjmp`/`longjmp` to insert `CPtrST`/`CPtrLD` instructions for those protected addresses. This way an attacker cannot use regular memory instructions to overwrite the return address and stack pointers in `jmp_buf`. The same approach can be applied to the C++ exception handling mechanism by instrumenting the appropriate APIs.

Context Switching. The permit-list contents (96 bytes) are maintained across context switches—as part of the process control block—if the process uses a permit-list. This step is likely to add minimal overhead (a few `LOAD` and `STORE` instructions takes $\leq 0.1\mu$S) to the OS context switch (typically $3-5\mu$S). Other OS-related tasks remain intact, such as inter-process data sharing, copy-on-write, and memory-mapped files.

[2]Only faulty store instructions are rejected to guarantee pointer integrity. We do not skip faulty loads as they do not change the control/data flow of the program.

Finally, as ZeRØ's metadata is inlined within the pointers themselves, they require no extra work for supporting multi-threaded applications.

VII. Security Analysis

In this section, we analyze the security guarantees provided by ZeRØ and its current limitations.

A. Security Discussion

Return Oriented Programming Attacks. Corrupting code pointers has been the most common and preferred attack vector over the last two decades. For instance, ROP attacks [41] and their just-in-time variant [45] typically start by corrupting the return address of a function to hijack the control flow of a program. ZeRØ's return address integrity effectively mitigates those attacks as it stops the adversary from leaking/overwriting return addresses using 1 bit of metadata per cache line in L2/L3 and main memory. For example, when an attacker tries to overflow a buffer to write to an adjacent return address, ZeRØ rejects the action and raises an advisory exception as the access violates the rules in Table I.

Jump- and Call-Oriented Programming Attacks. Protecting return addresses alone is not sufficient for more advanced attack variants. A variation of the ROP attack uses indirect branch instructions (`JMP`) to transfer control between gadgets. This attack technique is called jump-oriented programming (JOP) [5]. Another similar attack variant is call-oriented programming (COP) [16], which uses gadgets ending with an indirect `CALL` instruction. What makes JOP and COP similar is their use of code pointers for the indirect `JMP`/`CALL` instructions. As ZeRØ's code pointer integrity protects code pointers from being manipulated in memory, an attacker cannot use the pointers to launch a JOP/COP attack.

Counterfeit Object-Oriented Programming Attacks. Unlike ROP/JOP/COP attacks, which (re)use short instruction sequences, in counterfeit object-oriented programming (COOP) attacks, whole C++ functions are invoked through code pointers in read-only memory, such as `vtables` [39]. Each C++ object keeps a pointer (`vptr`) to its `vtable` (a table containing pointers to virtual methods). A method invocation, therefore, requires (a) dereferencing the `vtable` pointer, (b) computing the respective table index, and (c) executing an indirect `CALL` instruction with the table entry of the previous step as an operand. COOP attacks typically hijack program control-flow by overwriting `vptrs` of existent C++ objects and/or crafting counterfeit C++ objects with arbitrary `vptrs`. ZeRØ prevents COOP attacks by protecting code pointers inside the `vtables` and by using data pointer integrity to harden the `vptr` inside the C++ objects. For instance, using a regular `STORE` instruction to create a `vptr` will cause an advisory exception when the counterfeit `vptr` is accessed with a `DPtrLD` instruction.

Data-Oriented Programming Attacks. Unlike control-flow hijacking attacks, data-oriented programming (DOP) attacks do not alter the control flow of the program [20], [23], [36].

Instead, DOP attacks abuse data pointers to simulate the attacker's arbitrary computations using the original control flow of the victim program. Mitigating DOP has been a real challenge for prior defenses due the attack's use of data pointers. As data pointers are much more common than code pointers, the overheads of protecting them can be significant. ZeRØ's inlined metadata allows us to provide data pointer integrity with no performance cost. ZeRØ prevents those attacks by ensuring that regular `LOAD/STORE`s cannot corrupt data pointers.

Pointer Confusion Attacks. An attacker who has access to a `DPtrST` instruction may potentially overwrite any data pointer as all data pointers use the same encoding state (i.e., 11). To mitigate this issue, ZeRØ assigns a unique 10-bit identifier for every data pointer type and verifies it at `DPtrST`/`DPtrLD` call sites. This identifier prevents an attacker from using a vulnerable `DPtrST` instruction to corrupt arbitrary data pointers in memory. Instead, attackers will be restricted to accessing data pointers of the same (or compatible) type, thus reducing the attack surface. To comply with the C standard [22], ZeRØ permits accessing any data pointer using `void*` and `char*` without flagging a violation. All other data pointer types are considered incompatible. Similarly, ZeRØ mitigates code pointer confusion attacks by using the function type as a unique identifier at `CPtrST`/`CPtrLD` call sites. We report the total number of unique data- and code- pointer types for SPEC CPU2017 benchmarks in Section VIII.

Speculative Execution Attacks. Speculative execution attacks represent a major challenge for all modern security solutions [27]. They allow the attacker to leak program memory by first speculatively executing instructions that are not supposed to execute under normal conditions. The traces left behind in the microarchitecture by the speculatively executed instructions are then used to leak information covertly. While ZeRØ does not prevent speculative execution attacks, ZeRØ takes multiple steps to ensure speculative execution attacks cannot be used to bypass it. For example, a recent work (SpecROP [2]) shows that an attacker can speculatively chain multiple ROP gadgets. SpecROP uses speculative execution to prime the targets of indirect jump instructions and uses them to construct a gadget chain that leaks secrets. As all gadgets are speculatively executed, current defenses do not raise exceptions upon executing them. On the other hand, ZeRØ is resilient against SpecROP as we do not forward regular data (i.e., has a 00 state) to protected addresses (e.g., code pointers with a 10 state) in the processor pipeline. Instead, we mark those cases as potential violations and only raise our advisory exception when they become non-speculative. Thus, SpecROP gadgets will not be able to receive the attacker's primed targets.

In addition, it has been shown that speculative execution attacks can bypass ARM PAC by speculatively executing pointer signing instructions as gadgets to sign arbitrary pointers [17]. Once the pointers are signed, an attacker can leak the signature via a covert channel and use it to create a forged pointer. This forged pointer is then used to bypass ARM PAC authentication. This raises the question of whether an attacker can use a speculative execution attack to bypass ZeRØ. The short answer is No. Speculatively executing `CPtrLD`/`DPtrLD` instructions can only leak the pointer value. Leaking code and data pointers cannot alter the control/data flow of the program. On the other hand, overwriting the pointer requires a `STORE` instruction, which cannot be speculatively executed. Finally, `ClearMeta` instructions cannot be speculatively used to clear the pointer metadata bits as they are treated similarly to `STORE` instructions.

B. Limitations

Non-pointer Data Corruption. The main focus of this work is preventing the corruption of different pointer classes. It is to say, ZeRØ does not prevent regular (non-pointer) data from being corrupted through program `LOAD/STORE`s. Non-pointer (aka non-control) data attacks that tamper with or leak security-sensitive memory are possible [9]. Defeating non-pointer data attacks requires full memory safety, which typically comes with significant memory and performance overheads. Similar to recent hardware-based solutions (e.g., IntelCET [21], ARM PAC [37], and Morpheus [14]), we opt to exclude pure data attacks to simplify our design and performance requirements.

Third-party Code. Similar to prior work [30], ZeRØ provides pointer integrity for instrumented code only. Third-party libraries cannot take advantage of ZeRØ without recompilation. To facilitate communication with unprotected third-party code, ZeRØ provides a couple of options. The first option is to add the starting address and size of the third-party code to the permit-list. This way `LOAD/STORE` instructions from the third-party code operate normally without generating advisory exceptions. The second option is to clear the code- and data-pointer metadata in memory regions that are shared with third-party code before invoking external libraries. This is done by recognizing external library calls at the compiler level and inserting `ClearMeta` instructions accordingly. This way regular `LOAD/STORE` instructions in uninstrumented libraries can access the passed pointers without raising exceptions. ZeRØ, however, never clears the return address metadata bits if they are set, as return address integrity requires no program recompilation and thus can be provided for third party libraries and legacy binaries.

Memory Aliasing. Two memory access instructions can access the same memory location using different types. For example, a C union with a pointer and an integer member can be accessed using both regular `STORE` and `DPtrST` instructions. To avoid raising false alarms, ZeRØ statically detects such occurrences at compile time and emits regular `STORE` instructions for all union accesses. Similarly, we emit regular `STORE`s for pointers that are "cast" to integers before being stored to memory. While emitting regular `STORE`s for potential pointers reduces the security coverage, we opt for this solution to eliminate any false positives even if such C idioms are uncommon.

TABLE II: Area, delay and power overheads of ZeRØ (GE represents gate equivalent).

ZeRØ	**Area (GE)**	**Delay (*ns*)**	**Power (*mW*)**
L1 Overheads	[+5.41%] 531,175	[+0.05%] 1.99	[+3.37%] 30.7
L2-to-L1 Transformation	299	1.45	0.04
L1-to-L2 Transformation	326	1.72	0.04

VIII. Evaluation

In this section, we first measure the hardware overheads of implementing ZeRØ. Then, we compare ZeRØ's performance against prior solutions using the SPEC CPU2017 benchmark suite.

A. Hardware Measurements

ZeRØ adds additional operations to the L1 data cache and the interface between the L1 and L2 caches. Qualitatively, the area overhead of ZeRØ's L1 metadata is 3.125% as it adds 2B per 64B. As the metadata lookup happens in parallel to the L1 data and tag accesses, ZeRØ should have no impact on the L1 access latency. We verify this hypothesis by implementing ZeRØ on top of a 32KB direct mapped L1 data cache. We synthesize the baseline L1 data cache and the ZeRØ modified cache with the Synopsys Design Compiler and the 45nm NangateOpenCell library. We use OpenRAM [19] to generate SRAMs for both the data/tag arrays in L1 data cache and the bit vector arrays for ZeRØ. We report our VLSI measurements results in Table II.

As expected, the overheads associated with the ZeRØ pointer integrity are minor in terms of delay (0.05%) and power consumption (3.37%). The latency of the L2-to-L1 transformation module is less than the L1 data cache latency. This small latency implies that our transformation module can be folded completely within the pipeline stages and will not impact the performance-critical cache line fill operation. On the other hand, the latency of the L1-to-L2 transformation module is slightly higher (1.72ns). This is acceptable as the spill operation is not on the processor critical path. Thus, adding one more cycle to cache line evictions will not impact program execution time. Finally, the area and power overheads of our transformation modules are negligible compared to the L1 data cache.

B. Software Performance

We use the VLSI measurements as a guideline for our software evaluation. Our VLSI measurements show that ZeRØ's hardware changes have no impact on L1/L2 access latency. Thus, no extra clock cycles are needed to perform our integrity operations. In terms of program instructions, ZeRØ's return address integrity does not add any special instructions. Instead, it extends the functionality of regular `CALL`/`RET` instructions. On the other hand, ZeRØ uses special instructions to access code and data pointers. We note that the `CPtrLD`/`CPtrST` and `DPtrST`/`DPtrST` instructions simply replace traditional loads and stores for code and data pointers, respectively. They do not require any extra registers. We insert `MOV` instructions to encode the pointer types into the upper 10 bits of the `CPtrLD`/`DPtrLD` destination register and `CPtrST`/`DPtrST` source register (all are 48-bit wide pointers). We report the total number of unique data/code pointer types in Table III. Finally, ZeRØ inserts `ClearMeta` instructions upon heap/stack memory deallocation to remove the tags from the code- and data pointers, if they exist. We emulate the overheads of the `ClearMeta` instructions by inserting dummy `STORE` instructions in the corresponding (deallocation) code segments.

TABLE III: Number of unique LLVM function pointer types (**FPtrType**) and data pointer types (**DPtrType**) for the SPEC CPU2017 benchmark suite.

Benchmark Name	**Number of CPtrTypes**	**Number of DPtrTypes**	**Benchmark Name**	**Number of CPtrTypes**	**Number of DPtrTypes**
perlbench	17	72	xalancbmk	837	703
gcc	78	451	x264	50	23
mcf	0	6	blender	566	705
namd	5	10	deepsjeng	0	2
parest	48	611	imagick	21	54
povray	31	148	leela	1	16
lbm	0	2	nab	0	19
omnetpp	298	133	xz	14	18

C. Comparison with Prior Work.

To demonstrate the need for implementing ZeRØ, we compare it against the state-of-the-art pointer integrity technique, ARM PAC using the SPEC CPU2017 workloads. Prior work [30] showed that ARM PAC can be used to enforce code- and data-pointer integrity. As ARM PAC is only available in certain Apple SoCs with no support for third party code at the time of writing, we use the same emulation methodology as used by Liljestrand et al. [30] to estimate the performance overheads. We write a LLVM/Clang compiler [29] pass to insert four exclusive-or (`xor`) operations to account for the 4 cycle latency introduced by the PAC instructions. In addition to ZeRØ, we run three different instrumentation configurations:

- *PAC-FPtr.* In this configuration, ARM pointer authentication is applied to function pointer usages (i.e., forward-edge protection). Our compiler pass inserts the dummy instructions whenever a function pointer is loaded from memory (to emulate code pointer authentication) or stored to memory (to emulate code pointer signing).
- *PAC-RET.* In this configuration, ARM pointer authentication is applied to return addresses (i.e., backward-edge protection). Our compiler pass inserts the dummy instructions when a `CALL` instruction is executed (to sign the return address before pushing it to the stack memory) and when a `RET` instruction is executed (to authenticate the return address after loading it from memory).
- *PAC-Full.* In this configuration, ARM pointer authentication is applied to return addresses, code pointers, and data pointers. In addition to the first two configurations, we instrument all data pointer `LOAD` and `STORE` instructions to insert the dummy PAC instructions.

Fig. 8: Performance overheads of ZeRØ and three different ARM PAC configurations for the SPEC CPU2017 benchmark suite.

Evaluation Setup. We run our experiments on a bare-metal Intel Skylake-based Xeon Gold 6126 processor running at 2.6GHz with RHEL Linux 7.5 (kernel 3.10). We use the SPEC CPU2017 benchmarks with `ref` inputs and run to completion. To minimize variability, each benchmark is executed 5 times and the average of the execution times is reported. We notice negligible variance between the different runs.

Performance Results. Figure 8 shows the runtime overhead of the different design approaches (all normalized to baseline execution with no defenses). As the name suggests, ZeRØ introduces 0% performance overheads on average with a maximum of 0.6%. The overhead of PAC-FPtr is 3% on average with a maximum of 53%. The overhead of PAC-RET is 6% on average with a maximum of 59%.

Protecting all code and data pointers with PAC-Full results in 14% performance overheads on average with a maximum of 75%, which in many situations is considered too costly for ARM PAC to be practically deployed for data pointer protection. AOS [25] reduces the performance cost of data pointer integrity by using bounds tables and on-chip caches instead of signing/authenticating every data pointer. AOS reports an average performance overhead of 8.4% on SPEC CPU2006 workloads (by running the first 3 billion instructions on the gem5 simulator [4]). ZeRØ reduces the pointer integrity costs to zero by using minimal L1 metadata and only a 0.2% memory overhead.

IX. Related Work

In Section VIII, we showed that ZeRØ has clear performance advantages compared to the state-of-the-art commercial solution (ARM PAC). In this section, we explore other memory corruption mitigations and discuss their benefits and differences. Table IV divides prior work into three groups: shadow stack-based systems, encryption-based solutions, and full memory safety techniques.

Shadow Stack-Based Techniques. A straightforward solution to gurantee the integrity of return addresses is to adopt a shadow call stack [8]. Every time a `CALL` instruction is executed, the return address is pushed to the regular stack and an additional memory instruction stores a copy of the return address to the shadow stack. When a function returns, the original return address is restored from the stack and compared against the shadow return address. If an attacker manipulates the return address while stored on the stack, a mismatch occurs as the shadow stack is not accessible by the attacker. For example, Intel Control-flow Enforcement Technology (CET) [21], [42] makes its shadow stack inaccessible to program loads and stores, while CFI CaRE protects the shadow stack using ARM TrustZone-M security extensions [33]. Similar to ZeRØ's return address integrity, shadow stacks can be applied to legacy binaries with no compiler modifications. However, shadow stacks add an extra memory access operation for every function call and return, increasing energy and memory overheads.

Unlike return addresses, code pointers are not accessed in pairs of `CALL`/`RET` instructions. As a result, shadow stack-based defenses require an additional component to protect code pointers (aka forward-edge transitions). For example, Intel CET adds a new `ENDBRANCH` instruction, which is placed at the entry of each basic block that can be invoked via an indirect branch. When an indirect forward branch occurs, the following instruction is expected to be an `ENDBRANCH`, otherwise an attack is assumed. On the other hand, CFI CaRE instruments binaries in a manner which removes all function calls and indirect branches and replaces them with dispatch instructions that trap control flow to a branch monitor. The branch monitor verifies the control-flow transition by comparing it against a pre-determined (i.e., compile time) control flow graph (CFG) of the program. Trapping into the branch monitor for every indirect call causes CFI CaRE's performance overheads to range between 13% and 513%. More importantly, techniques that rely on static analysis to construct a CFG and enforce it at runtime are ultimately limited by the precision of the analysis [7]. ZeRØ's simple instruction set extensions implicitly protect forward edge transitions by guaranteeing code pointer integrity with zero cost.

Code Pointer Integrity (CPI) [28] and its relaxed variant (Code Pointer Separation) use compiler analysis and instrumentation to isolate code pointers into a separate region of memory. The idea is similar to the concept of shadow stacks, but extends it to include code pointers in globals and heap objects. Unlike ZeRØ's inlined metadata, CPI requires extra memory accesses per every sensitive pointer access to fetch

TABLE IV: Comparison Against Prior Works. The assets protected by each system is listed: **RET** stands for return address integrity/protection, **CPtr** stands for function pointer integrity, **DPtr** stands for data pointer integrity, and **Data** stands for non-pointer data integrity.

Proposal	Protected Assets				Main Operations	Hardware Changes	Metadata Overhead
	RET	CPtr	DPtr	Data			
Intel CET [21]	✓	✗	✗	✗	Memory access per CALL/RET and EndBranch Check	Isolated shadow stack and indirect branch FSM tracker	1 word per return address
CFI CaRE [33]	✓	✓	✗	✗	Memory access per CALL/RET and Branch verification	TrustZone-protected shadow stack	1 word per return address
CPI [28]	✓	✓	✗	✗	Extra memory accesses per every sensitive pointer access	No changes	4 words per sensitive pointer
Branch Regulation [24]	✓	✓	✗	✗	Memory access per CALL/RET and Branch verification	Isolated Secure Call Stack and a function bounds cache	3 words per stack frame
CCFI [31]	✓	✓	✗	✗	Pointer signing/authentication	AES co-processor	ptr-inlined
ARM PAC [37], [30]	✓	✓	✓	✗	Pointer signing/authentication	QARMA co-processor	ptr-inlined
HDFI [46]	✓	✓	✗	✗	Tag check	1B per L1D line, Tag$, and DFITagger unit	1 bit per word
Morpheus [14]	✓	✓	✓	✗	Pointer encryption/decryption and pointer displacement	QARMA co-processor, Tag$, and churn unit	2 bits per word
Intel MPX [34]	✓	✓	✓	✓	2+ mem ref for bounds	Unknown (closed platform)	2 words per ptr
CHERI [52]	✓	✓	✓	✓	1+ mem ref for capability and capability management instructions	Capability coprocessor, Tag$ and Capability Unit	Ptr size is 4x
CHEx86 [43]	✓	✓	✓	✓	1+ mem ref for capability and pointer tracking	μop injection logic, Capability$ alias$, and speculative pointer tracker	2 words per ptr
MemTracker [51]	✓	✓	✓	✓	1+ mem ref for state checks and states updates	Programmable table, 32 instructions, state$, and 2 extra pipeline stages	n bits per word
Califorms [38]	✓	✓	✓	✓	Tag check	8B per L1D line, 1 bit per L2 line	1-7B per ptr
ZeRØ	✓	✓	✓	✗	Tag check	2B per L1D line, 1 bit per L2 line	ptr-inlined

its corresponding metadata. Moreover, prior work showed that CPI's safe region can be leaked and then maliciously modified by using data pointer overwrites, undermining the security guarantees of the solution [13].

Encryption-Based Techniques. To eliminate the memory costs associated with shadow stacks, prior work used encryption to randomize the pointer layout before storing it to memory. As long as the attackers have no access to the encryption key, they cannot reliably leak/overwrite the pointer. Early work used XOR-based encryption to avoid adding performance costs to every pointer load/store operation [10], [50]. As XOR-based encryption is vulnerable to known plaintext attacks, modern work utilizes strong encryption, such as AES in cryptographic control-flow integrity (CCFI) [31] and QARMA ciphers in ARM PAC [37], [30] and Morpheus [14]. Our software evaluation shows that ZeRØ completely eliminates the runtime overheads associated with ARM PAC for code and data pointer protections.

Another example of an encryption-based defense is Morpheus [14], an architecture that (i) displaces code and data pointers in the address space (ii) diversifies the representation of code and pointers using strong encryption, and (iii) periodically repeats the above steps using a different displacement and key. Similar to ZeRØ, Morpheus does not protect non-pointer data corruption and provides low performance overheads. Unlike ZeRØ which is a secret-less solution, Morpheus must keep two parameters secret until they are changed: displacements for the code and data regions, and keys for encrypting/decrypting pointers. Additionally, a key limitation of encryption-based techniques is the additional energy costs per pointer operation. One AES operation can cost up to 48.02pJ/bit (or 3073.28pJ per 64-bit encryption) at 1 MHz while one QARMA operation costs 7.78pJ/bit (or 497.92pJ per 64-bit encryption) [15], which is at least an order of magnitude higher than ZeRØ's 2-bit metadata read and check in the L1 data cache (energy consumption of L1 data access ranges between 64 and 105pJ/Byte, i.e., 16 and 26.25pJ per two bits [32]).

Memory Safety Techniques. Since memory safety vulnerabilities are the root cause of the majority of memory attacks, researchers and manufacturers have proposed many hardware solutions to address this problem, including base and bounds techniques [11], [34], [53], [43], [49], memory tagging [35], [1] and tripwires [44], [38]. While the aforementioned techniques protect both pointer and non-pointer data items, the biggest hurdle for adopting them is their performance overheads. For example, Intel MPX can introduce up to 4x performance overheads [34], whereas a recent capability-based system, CHEx86, introduces 14% runtime overheads and 38% memory overheads [43]. ZeRØ instead provides code and data pointer integrity, which is sufficient to prevent a wide range of code reuse and data-oriented programming attacks, at no runtime cost.

While state-of-the-art tripwires systems (i.e., REST [44] and Califorms [38]) come with low performance overheads, both are vulnerable to non adjacent buffer overflow attacks.

For example, an attacker can leverage a non adjacent buffer overflow to jump over the redzones of REST/Califorms and corrupt the victim pointer. ZeRØ is resilient against non adjacent buffer overflows, which represent 27% of Microsoft's memory safety CVEs [3]. Finally, while changing the cache line format between L1 and L2 was first introduced in Califorms [38], ZeRØ uses a simpler encoding that reduces the complexity of the L1/L2 transformation modules and avoids adding any latency to the performance-critical fill operation. Moreover, Califorms' metadata is used to deny accesses to dead bytes whereas ZeRØ's metadata is used to enforce access control rules on neighboring data (i.e., the rest of bytes in the code/data pointer).

X. Conclusion

Most end users want security but do not want the inconvenience of having it: they do not want their batteries drained, or apps slowed, or to be bothered with updates and crashes. This is the unfortunate reality that sends novel security techniques with even minor performance overheads to the crypt of great security ideas. Techniques that have been mass deployed in hardware (e.g., WˆX and SMEP/SMAP) are the ones that have close to zero overheads. Even techniques like ARM's Pointer Authentication (PAC) —which does have significant overhead when applied fully—is applied partially to only protect code pointers, and only to the kernel to keep the overheads small. Thus, low performance overhead and convenience are key to widespread adoption of security techniques.

In this paper we proposed ZeRØ, a hardware primitive for resilient operation under memory corruption attacks with zero overhead. ZeRØ enforces code and data pointer integrity with minimal metadata. Specifically, using 1 bit per 64 bytes in L2 and beyond, and 3.125% area overhead in the L1, ZeRØ is able to protect the integrity of both code and data pointers. As a result, ZeRØ incurs 0% performance degradation compared to 14% for the state-of-the-art ARM PAC when applied to its full extent. ZeRØ matches or offers better security guarantees than ARM's PAC and Intel's CET. Moreover, our VLSI results showed that ZeRØ can be implemented with minimal latency, area, and power overheads.

The techniques described in the paper offer exploit mitigation at no cost and are a perfect complement to systems that identify and mitigate a broader class of memory attacks, such as No-FAT [49]. Extant memory safety techniques are more suitable for testing before apps are distributed to customers where higher overheads can be tolerated, while exploit mitigation techniques such as ZeRØ which offer no overheads and resilient operation are more suitable for end user deployment. This will at least be the case until memory safety techniques can be offered with 0% runtime overhead.

Acknowledgment

This work was partially supported by FA8750-20-C-0210, a Qualcomm Innovation Fellowship, and a gift from Bloomberg. Any opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US government or commercial entities. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc.

References

[1] ARM, "Memory tagging extension: Enhancing memory safety through architecture," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety, 2019.

[2] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *RAID '20: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, San Sebastian, Spain, October 2020, pp. 1–16.

[3] J. Bialek, K. Johnson, M. Miller, and T. Chen, "Security analysis of memory tagging," 2020. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ASIACCS '11: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, March 2011, pp. 30–40.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to RISC," in *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 2008, pp. 27–38.

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[8] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *SP '19: Proceedings of the 2019 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2019, pp. 985–999.

[9] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, Baltimore, MD, USA, 2005.

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: Protecting pointers from buffer overflow vulnerabilities," in *SSYM '03: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, Washington, DC, USA, 2003.

[11] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "HardBound: architectural support for spatial safety of the C programming language," in *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[12] U. Drepper, "Security enhancements in redhat enterprise Linux (beside SELinux)," 2005. [Online]. Available: https://akkadia.org/drepper/nonselsec.pdf

[13] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, 2015, pp. 781–796.

[14] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco, S. Malik, M. Tiwari, and T. Austin, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, 2019, pp. 469–484.

[15] S. J. Ghangro, "Block ciphers for low energy," Ph.D. dissertation, KU Leuven, July 2017. [Online]. Available: https://www.esat.kuleuven.be/cosic/publications/thesis-293.pdf

[16] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *SP '14: Proceedings of the 2014 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2014, pp. 575–589.

[17] GoogleProjectZero, "Examining pointer authentication on the iphone XS," 2019. [Online]. Available: https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

[18] R. H. Gumpertz, "Combining tags with error codes," in *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, 1983, pp. 160–165.

[19] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *ICCAD '16: Proceedings of the 35th International Conference on Computer-Aided Design*, Austin, TX, USA, 2016.

[20] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *SP '16: Proceedings of the 2016 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2016, pp. 969–986.

[21] Intel, "Intel control-flow enforcement technology preview," 2017. [Online]. Available: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[22] International standardization working group for the programming language C, "ISO/IEC 9899:202x. ISO/IEC," http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2346.pdf, 2019, [Online; accessed 01-May-2021].

[23] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Toronto, Canada, 2018, pp. 1868–1882.

[24] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *ISCA '12: Proceedings of the 39th Annual International Symposium on Computer Architecture*, Portland, OR, USA, 2012, pp. 94–105.

[25] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *MICRO-53: Proceedings of the 53nd Annual IEEE/ACM International Symposium on Microarchitecture*, Global Online Event, 2020, pp. 1153–1166.

[26] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. 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 *ISCA '14: Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014, p. 361–372.

[27] P. Kocher, 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 *SP '19: Proceedings of the 40th IEEE Symposium on Security and Privacy*, May 2019.

[28] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI'14: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 147–163.

[29] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis transformation," in *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2004, pp. 75–86.

[30] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, August 2019, pp. 177–194.

[31] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *CCS '15: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, Denver, Colorado, USA, 2015, pp. 941–951.

[32] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86_64 processors," in *Proceedings of the International Conference on Green Computing*, Chicago, IL, USA, 2010, pp. 123–133.

[33] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *RAID '17: Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, 2017, pp. 259–284.

[34] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel mpx explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, p. 28, 2018.

[35] Oracle, "Hardware-assisted checking using silicon secured memory (SSM)," 2015. [Online]. Available: https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html

[36] J. Pewny, P. Koppe, and T. Holz, "STEROIDS for DOPed applications: A compiler for automated data-oriented programming," in *EuroS&P '19: Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, Stockholm, Sweden, June 2019.

[37] Qualcomm Technologies Inc, "Pointer authentication on ARMv8.3," 2017. [Online]. Available: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[38] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using Califorms," in *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, October 2019, pp. 558–571.

[39] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2015, pp. 745–762.

[40] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *arXiv.org*, February 2018.

[41] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2007, pp. 552–561.

[42] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *HASP '19: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, Phoenix, AZ, USA, June 2019.

[43] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *ISCA '20: Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, Valencia, Spain, June 2020, pp. 762–775.

[44] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *ISCA '18: Proceedings of the 45th Annual International Symposium on Computer Architecture*, Los Angeles, CA, USA, 2018, pp. 600–611.

[45] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *SP '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, May 2013, pp. 574–588.

[46] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *SP '16: Proceedings of the 2016 IEEE Symposium on Security and Privacy*, May 2016, pp. 1–17.

[47] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *SP '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2013, pp. 48–62.

[48] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *SEC '17: Proceedings of the 26th USENIX Conference on Security Symposium*, Vancouver, BC, Canada, 2017, pp. 1057–1074.

[49] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-FAT: Architectural support for low overhead memory safety checks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, June 2021.

[50] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, Portland, OR, USA, 2004, pp. 209–220.

[51] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," in *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, USA, 2007, pp. 273–284.

[52] 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 *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*, San Jose, CA, USA, May 2015, pp. 20–37.

[53] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, , A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. Moore, "CHERI concentrate: practical compressed capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, October 2019.