No-FAT: Architectural Support for Low Overhead Memory Safety Checks

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

***Abstract*—Memory safety continues to be a significant software reliability and security problem, and low overhead and low complexity hardware solutions have eluded computer designers. In this paper, we explore a pathway to deployable memory safety defenses. Our technique builds on a recent trend in software: the usage of binning memory allocators. We observe that if memory allocation sizes (e.g., malloc sizes) are made an architectural feature, then it is possible to overcome many of the thorny issues with traditional approaches to memory safety such as compatibility with unsecured software and significant performance degradation. We show that our architecture, No-FAT, incurs an overhead of 8% on SPEC CPU2017 benchmarks, and our VLSI measurements show low power and area overheads. Finally, as No-FAT's hardware is aware of the memory allocation sizes, it effectively mitigates certain speculative attacks (e.g., Spectre-V1) with no additional cost. When our solution is used for pre-deployment fuzz testing it can improve fuzz testing bandwidth by an order of magnitude compared to state-of-the-art approaches.**

***Index Terms*—Bounds Checking, Fuzzing, Memory Safety, Microarchitecture, Spectre-V1, Systems Security.**

I. Introduction

Memory safety violations in programs have provided a significant opportunity for exploitation by attackers. For instance, Microsoft recently revealed that the root cause of around 70% of all exploits targeting their products are software memory safety violations [37]. Similarly, the Project Zero team at Google reports that memory corruption issues are the root-cause of 68% of listed CVEs for zero-day vulnerabilities between 2014 and 2019 [22].

To address the threat of memory safety, software checking tools (e.g., AddressSanitizer [52]) and fuzz testing are widely deployed. In software fuzz testing, binaries are instrumented with a tool like AddressSanitizer to detect memory safety vulnerabilities and run with inputs mutated from a set of exemplary inputs in the hopes of detecting bugs before deployment. Google has reported that it has been fuzzing about 25,000 machines continuously since 2016, which has resulted in the identification of many critical bugs in software such as Google Chrome and several open source projects [7]. Assuming 15 cents per CPU hour for large memory machines—a requirement for reasonable performance on fuzz testing—the investment in software fuzzing for detecting memory errors could be close to a billion dollars at just one company.

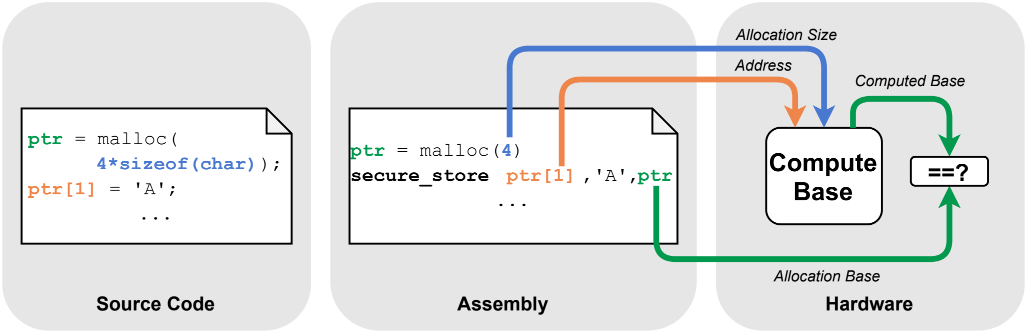
Despite a Herculean effort by software vendors, memory safety vulnerabilities continue to slip through, ending up in deployed systems. Recognizing that pre-deployment fuzz tests can never be complete, companies have also proposed post-deployment crowdsourced fuzz testing [38], [59]. For instance, Mozilla recently created a framework for fuzzing software using a cluster of fuzzers made by users who are willing to trade and contribute their CPU resources (e.g., using office workstations after-hours for fuzz testing) [38]. Assuming that many companies participate and these tests run for enough time, on a global scale, the amount of energy invested in producing reliable software may be even higher than the amount of time running the software with crowdsourced testing. Thus, increasing the efficiency of memory error detection can have significant green benefits in addition to improving security and reliability.

Researchers and commercial vendors have also stepped up to the call to reduce inefficiencies in software testing and security. There is a long history of academic proposals that have continuously chipped away at these overheads for detecting memory safety vulnerabilities over the past 25 years ([51], [3], [47], [26], [14], [39], [61], [56], [50], [55], [27]). Commercial vendors have also proposed or manufactured hardware with support to mitigate these overheads (Intel's MPX [43], ARM's MTE [2], and Oracle's ADI [45]).

In this paper, we show that the overheads of providing memory safety can be decreased even further with novel hardware support. Traditional memory safety techniques incur overheads from two sources: (1) storage of metadata to detect memory safety violations, and (2) computational overheads of memory safety checks based on the stored metadata. No-FAT[1], our system, eliminates all metadata attached to pointers, and hides the computational overheads of the metadata checks by performing them in parallel with regular lookups.

The technological change that facilitates these improvements in No-FAT is the increasing adoption of binning allocators. Binning memory allocators use collections of pages (called bins), where each bin is used to allocate objects of the same size. Using bins enables the allocator to quickly serve

[1] The name is an allusion to No-Fat Milk, which has fewer calories. Also, closely related work in this area refer to their schemes as Fat and Low Fat pointers.



```
ptr = malloc(
      4*sizeof(char));
ptr[1] = 'A';
         ...
```

```
ptr = malloc(4)
secure_store  ptr[1] ,'A',ptr
          ...
```

Fig. 1: A high level overview of how No-FAT makes allocation size an architectural feature.

allocation requests and increases performance by maintaining allocation locality [21], [20], [18], [34].

No-FAT, when used with a binning allocator, is able to *implicitly* derive allocations bounds information (i.e., the base address and size) from the pointer itself without relying on explicit metadata. The hardware/software contract has to be tweaked slightly to facilitate No-FAT and binning allocators working together: the standard allocation sizes used by a binning allocator need to be supplied to the hardware and special load and store instructions are created to access the allocation sizes. In other words, the memory allocation size (e.g., malloc size) becomes an architectural feature.

We illustrate one type of protection offered by No-FAT with an example shown in Figure 1. The program allocates an array of four characters and writes `'A'` to the second element, `ptr[1]`. To make the allocation an architectural feature, No-FAT modifies the compiler to propagate the allocation base address (i.e., `ptr`) to memory instructions (i.e., `secure_store`). Before accessing memory, No-FAT computes the allocation base of the memory address (i.e., `ptr[1]`). No-FAT will then compare the computed base address against the compiler propagated base address (i.e., `ptr`) and raise an exception in the case of a mismatch. Moreover, No-FAT enforces temporal protection by generating a random tag upon memory allocation and storing it in the currently unused upper bits of the pointer. Then, upon executing memory instructions, No-FAT verifies that tags match.

No-FAT also provides resilience against a subset of speculative execution attacks, namely Spectre-V1 [29]. Spectre-V1 attacks exploit speculative execution to access out-of-bounds memory, effectively bypassing software-based bounds checks. No-FAT's memory instructions are aware of allocation bounds information. Thus, allocation bounds information can be used to verify if memory accesses are within valid bounds even for speculatively executed instructions.

A challenge with prior memory safety proposals, or for that matter any proposal that involves ISA changes, is practical deployment considerations. Are the new instructions and interfaces compatible with older software? Will there be performance degradation when using older software? Does code have to re-written or can it be simply recompiled? Luckily, the key ideas described in this paper viz., the idea of standardizing memory allocation structures, and using that information to provide memory safety, have been very well tested with software implementations [1], [16], [19]. Moreover, No-FAT has three advantages over prior works. First, prior works suffer from high performance overheads (100%) which this work mitigates. Second, we show that a degree of temporal safety and intra-object spatial safety can be offered by our implementation over prior software works with simple modifications. Third, we improve over prior works by providing support for arbitrary sized allocations (as opposed to power-of-two allocation sizes).

All of No-FAT's software transformations are performed using the Clang/LLVM compiler framework [32]. Our experimental results with the SPEC CPU2017 benchmark suite indicate that the overheads of No-FAT are on average 8% with very conservative measurements. Our VLSI implementation results with 45nm NangateOpenCell show that No-FAT can be efficiently added to modern processors with negligible performance, area, and power overheads.

In summary, this paper makes the case for standardizing memory allocation sizes and explicitly making this information available to the architecture. We observe at least three distinct benefits:

- **Improving fuzz-testing time.** Currently companies spend hundreds of millions of dollars testing software programs for bugs. A majority of these bugs tend to be intricate memory safety bugs. Exposing allocation sizes to hardware simplifies the checks for memory safety and improves the fuzz testing bandwidth by over 10x based on state-of-the-art solutions (e.g., AddressSanitizer based fuzzing).
- **Improving run-time security.** Despite the best effort of software engineers to produce bug-free code, some of these bugs do end up in production, and pose a risk to end users. If users wish to protect against remaining residual risk, our solution offers the lowest overhead protection among all published memory safety solutions that thwart data corruption attacks.
- **Improving resilience to Spectre-V1 attacks.** Exposing allocation sizes to the hardware allows the hardware to effectively perform bounds checking even for speculative memory accesses.

II. No-FAT System Overview

A. Preliminaries

Binning memory allocators have gained prominence in the past decade and are now widely used [21], [20], [18], [34]. In a binning alloctor, the heap is divided into regions where each region is used to allocate objects of a pre-determined size. Thus, the memory size returned to a program is rounded up to one of the standard sizes offered by the allocator. For example, allocation requests that are less than 16 bytes come from the first region, allocation requests for 16 to 32 bytes come from the second region and so on. In contrast, non-binning allocators can provide the exact amount of memory requested by the program at the cost of an additional allocation header to store its size [33]. Binning allocators trade off a little memory fragmentation for faster allocation and deallocation times, and practically speaking, the fragmentation overheads tend to be negligible for most programs. In this paper, we expose the pre-determined sizes offered by a binning memory allocator to the hardware to provide memory safety.[2]

B. How does No-FAT provide Inter-allocation Spatial Memory Safety?

The goal of inter-allocation spatial memory safety is to be able to identify pointer-based accesses that access addresses outside the region of memory allocated to that pointer. To perform this check we need three pieces of information: (1) the starting address of the allocation, (2) the size of the space allocated to the pointer, and (3) the address of the pointer-based access. The benefit of binning allocators is that (1) and (2) can be computed from (3) using simple arithmetic and concurrently with the data access.

Given a pointer address, we determine the region that the pointer is from. Say each region is S GiB, and the heap starts at address H. Then the region of the pointer is $(ptr - H) >> log_2(S)$. Once the region is known, we can know the size of the allocation because all allocations from the region are of the same size. The base address of the allocation can be computed by $\lfloor(ptr/size)\rfloor * size$, whereas the combination of integer division and multiplication has the effect of rounding ptr down to the nearest $size(ptr)$-aligned boundary, which is the base address.

For example, let us assume that the heap starting address is $H =$ `0x380000000000` and the memory allocator uses 64 bins (i.e., regions) each of size 32 GiB, where the third region is used to store allocations of size 32B. When the program executes `char* A = malloc(32)`, the memory allocator might return the following base address: `0x381000000040`. Now, given an arbitrary pointer $ptr =$ `0x381000000045`, the hardware computes the region number by subtracting the heap starting address (i.e., `0x001000000045`) and ignoring the 35 LSBs (i.e., `0x002`, which is the third region). Then, the hardware retrieves the allocation size from the hardware table. Finally, the base address can be computed as $\lfloor(\texttt{0x381000000045}/32)\rfloor * 32 = \texttt{0x381000000040}$.

How does this information help protect against attacks? Let us say an attacker has the ability to control the index variable of a dynamically allocated array. With this ability, the attacker can cause the pointer to go out-of-bounds and subvert the memory instruction in order to read/write from a different allocation. If we simply calculated the base address from the attacker modified address we would not be able to catch the attack since we do not have an expectation of what the base address ought to have been. To avoid this case, No-FAT extends memory access instructions with an extra operand that carries a trusted base address. The trusted base address is simply the base address returned by `malloc`. This way we can verify the correctness of the access by computing the base address of the input pointer and matching it against the trusted base address, which is part of the instruction.

Computing the base address of a pointer for every memory access instruction is a costly operation as it includes a 64-bit division operation followed by a 64-bit multiplication. Division is a relatively expensive operation even on modern CPUs. To simplify the bounds checking operation, No-FAT uses the following check $isValid(ptr, base) = ptr - base < size(base)$. The idea is simple. As the instruction holds the trusted base address, we first compute its corresponding size by extracting the region number as explained before. Then, we compare this size to the difference between the input pointer and the trusted (i.e., instruction-based) base address. If the pointer overflows to an adjacent allocation, the difference will be larger than the computed difference. If the pointer underflows to a previous allocation, $ptr - base$ will be a negative number that will be interpreted as a large positive number that is $\geq size(base)$ as we use unsigned arithmetic.

To make No-FAT compatible with unprotected code, memory instructions that need to perform the check are emitted using special instructions. Specifically No-FAT uses Secure Load (`secure_load`) and Secure Store (`secure_store`) instructions (see Section III) that use the allocation base address as a distinct operand. This operand is propagated in the binary using a compiler pass (see Section V). This way `secure_load` and `secure_store` can verify access boundaries using the *isValid* check, as described above. On machines which do not have hardware support for No-FAT, `secure_load` can be interpreted as a regular load and the third operand will be ignored.

C. How does No-FAT provide Intra-allocation Spatial Memory Safety?

The goal of intra-allocation spatial memory safety is to prevent overflows from one field to another within the same allocation. The strategy used by No-FAT for intra-allocation safety is to convert the intra-allocation memory safety problem to an inter-allocation problem. No-FAT uses a source-to-source transformation, Buf2Ptr, which has been previously used in the area of data layout optimizations for enhancing performance [25], [49], [66]. Buf2Ptr promotes buffer fields, which

[2] A recent study [60] proposes passing semantic information from software to hardware to achieve better resource utilization and enhance performance. However, neither allocation size nor fine-grained security were included.

exist in C/C++ structs (or classes), into their own allocations. To illustrate Buf2Ptr, consider the example in Listing 2. The array field, `buf[10]`, within the struct, `Foo`, is replaced with a promoted pointer, `p_buf`, and a new variable for the original array is defined (`Foo_buf[10]`). As a result of this transformation, allocations, deallocations, and usages of the original field must also be properly promoted. For example, an allocation for a composite data type (e.g., `Foo`) becomes separate allocations based on the number of fields promoted (e.g., `Foo_buf`). As the standalone allocations have their own base address, they can be protected with No-FAT, as described above.

```

struct Foo {
  char buf[10];
  int value;
};


struct Foo *f = malloc(
  sizeof(struct Foo));



f->buf[7] = 'A';

free(f);

```

(a) Original

```
// Promoted Type
char Foo_buf[10];
struct Foo {
  char *p_buf;
  int value;
};
// Promoted Allocations
struct Foo *f = malloc(
  sizeof(struct Foo));
f->p_buf = malloc(
  sizeof(Foo_buf));
// Promoted Usages
f->p_buf->buf[7] = 'A';
// Promoted Deallocations
free(f->p_buf);
free(f);
```

(b) Transformed

Listing 2: An example of Buf2Ptr transformation.

D. *Temporal Memory Safety.*

To enforce temporal memory safety, No-FAT tags the upper 16-bits of data pointers on 64-bit systems with a random value upon `malloc`. This value is propagated in our new instructions (i.e., `secure_load` and `secure_store`) as part of the memory address and the allocation base address. This way, comparing the tag of the memory address with the tag of the allocation base address catches temporal safety violation with a probability of $1-(1/2^{16}) = 99.9984\%$. When a virtual memory region is reallocated to a different object it receives a new random tag, implicitly nullifying all dangling pointers which used to point to the old object, as they are likely to have different tags.

E. *Handling Procedure Calls and Nested Pointers*

Consider the following: `q = p + 16; x = Bar(q);`. Here, `p` is a pointer to a 32B allocation, and `q` is a derived pointer to a field within the allocation. In this case, the use of a pointer happens in a different function (aka context) than the one where it was originally created. Thus, all functions using the base pointer (i.e., `p`) or its derivatives (e.g., `q`) need to be given access to the base address. One way to do this would use a source-to-source transformation to add an extra operand to all functions that use pointer arguments. This way the address would be in the stack of the needed function. This solution requires changing the function signature and breaks compatibility with unprotected code.

Instead, we use a different, simpler abstraction. Whenever a data pointer goes out of context (i.e., passed to another function or spilled to memory), we first verify that it is an in-bounds pointer using a Verify Bounds (`verify_bounds`) instruction (see Section III). When a pointer is loaded from memory, we first compute its base address using a `compute_base` instruction and propagate this base address to all the following memory instructions as a third operand.

With this approach, can the attacker abuse pointers that escape to another function? This is not possible because (1) we verify the bounds of the pointer before spilling it to memory and (2) we protect the memory with No-FAT so we are assured that the pointer stored in memory cannot be overwritten. This abstraction also permits No-FAT to use only intra-procedural analysis, which simplifies the implementation. Going back to our example, we first verify the bounds of `q` before calling `Bar(q)`. This is done with one `verify_bounds` instruction that takes the base address of `q` as an operand and matches it against the computed base address of `p + 16`. Inside `Bar`, we first call `compute_base` with `q` as an operand to retrieve its base address and propagate this base address to all memory instructions that uses `q` as an address.

III. Architecture Support

No-FAT adds the following instructions to the ISA:

- **`secure_store/secure_load <R1>, <R2>, <R3>`**: These instructions use three register operands. The values in registers `R1` and `R2` point to the store/load address and source/destination register as usual. The value in register `R3` is reserved for the allocation base address and is propagated by the compiler. Upon executing this instruction, the hardware computes the allocation size of `R3` and compares it to the difference between the address stored in `R1` and `R3`. An exception is thrown in case of $\mathtt{R1}-\mathtt{R3} \geq \mathit{size}(\mathtt{R3})$. Additionally, the hardware matches the upper 16 bits of `R1` and `R3` to detect temporal memory safety violations.
- **`verify_bounds <R1>, <R2>`**: This instruction is used to check the bounds of pointers before storing them to memory (or passing them to a different function). It uses two register operands. The value in register `R1` is a pointer whereas the value in register `R2` is reserved for the allocation base address and is propagated by the compiler. Similar to `secure_store` and `secure_load`, upon executing this instruction, the hardware computes the allocation size of `R2` and compares it to the difference between the address stored in `R1` and `R2`. An exception is thrown in case of $\mathtt{R1}-\mathtt{R2} \geq \mathit{size}(\mathtt{R2})$ to indicate that an out-of-bounds pointer is being stored to memory.
- **`compute_base <R1>, <R2>`**: This instruction takes a memory address (i.e., pointer) as input in `R1` and returns the allocation base address of this pointer in `R2`. This instruction is used to retrieve the correct base address of pointers that are passed to different contexts (e.g., through function calls).

IV. Microarchitecture Design

In this section, we describe the four hardware components that are needed to enable No-FAT.

MAST. The Memory Allocation Size Table is a hardware structure, which is initialized at program startup with a process's allocation size configuration. The table is designed to work with binning allocators. The MAST enables No-FAT to support generic (i.e., non-powers-of-two) allocation sizes for each bin.[3]

In this work, we use a simple binning allocator that divides the heap into N equally sized bins. Based on our experiments, using 64 distinct bins is sufficient to balance performance and memory utilization. Thus, we use a 64-entry MAST with an entry size of 16B resulting in a total size of 1KB. Each entry holds an 8B size field and an 8B inverse size field. The size field of the nth entry is used to hold the allocation size used for the nth allocator bin. The inverse size field is an optimization that is discussed later. As a program's heap is contiguous, we use a single hardware register to store the starting address of the program heap and use it to derive the starting address of all bins. Some binning allocators (e.g., TCmalloc [21] and Jemalloc [20]) may change the allocation size used by one bin at runtime if all objects in the bin are freed. In this case, the allocator can simply update the MAST entry with the new size. We leave the investigation of other memory allocators to future work.

Bounds Checking Module. The bounds checking module takes two 64-bit operands, Ptr and $Base_{Ptr}$. It subtracts the two operands and compares the result with the allocation size of $Base_{Ptr}$. To compute the size of a given base pointer, the bounds checking module first maps the pointer to an allocation bin using simple subtract and shift operations followed by an access to the MAST to retrieve the allocation size. Next, the bounds checking module uses a subtraction operation ($Ptr - Base_{Ptr}$) followed by a 64-bit unsigned comparison with the recently retrieved size (i.e., $size(Base_{Ptr})$). The last step is the temporal check, which is done with a 16-bit comparison operation between the upper 16 bits of Ptr and $Base_{Ptr}$.

The bounds checking module is invoked during the secure_load and secure_store instructions to prevent out-of-bounds pointer dereference and during the verify_bounds instruction to prevent out-of-bounds pointers from escaping to memory. As shown in Figure 2, the check operation can be totally hidden within the access latency for the L1 data cache.

Base Computing Module. As discussed in Section II-E, pointers can be passed from one context to another. As No-FAT relies on simple intra-procedural compiler analysis, it needs to recompute the base address every time a pointer is loaded from memory (e.g., double pointers) or used as a function argument. This feature is currently implemented with compute_base instruction that invokes the Base Computing Module.

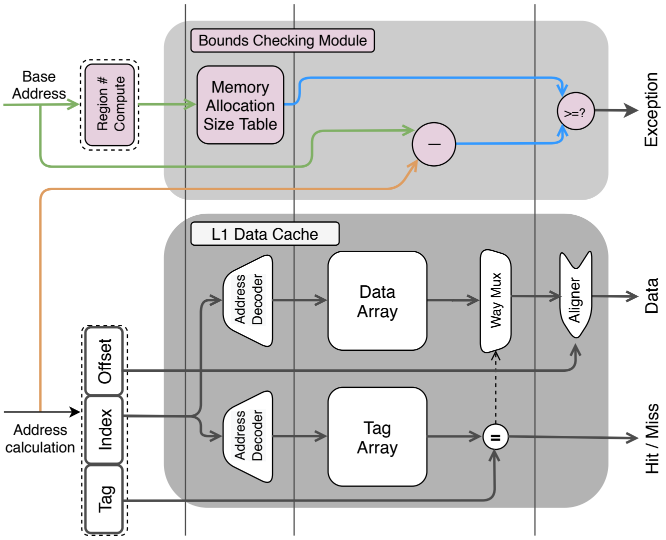


Fig. 2: Pipeline diagram for the L1 cache hit operation. The bounds checking operations (top) are pipelined to avoid adding any access latency to L1 data.

This module takes a 64-bit ptr operand and computes its base address using $\lfloor ptr/size(ptr) \rfloor * size(ptr)$. While $size(ptr)$ requires one MAST lookup, the division operation is costly. No-FAT uses a common optimization that replaces the expensive division ($ptr/size(ptr)$) with a cheaper multiplication ($ptr * (1/size(ptr))$) by using fixed-point arithmetic. This approach is feasible since the set of allocation sizes is constant, and thus the set of allocation size reciprocals can be pre-calculated and stored in the MAST along side with allocation size.

Dedicated Register File. As our secure_load and secure_store instructions use a third register operand, they may introduce register pressure. Thus, No-FAT adds a set of architectural registers that the compiler can exclusively use for holding and propagating allocation base addresses. The new registers are saved in a separate register file that is accessed in parallel to the regular register file.

V. Software Design

In this section, we describe the memory allocator, compiler and operating system changes to support No-FAT.

A. Dynamic Memory Management

One of No-FAT's key contributions is making the allocation size an architectural feature (i.e., sharing the allocation size information between software and hardware). To enable this feature, No-FAT requires binning memory allocators, in which a memory page is used to allocate objects of the same size. No-FAT does not add any constraints on how the allocator manages its internal metadata (e.g., free lists). No-FAT only intercepts calls to common memory management operations. For example, No-FAT intercepts all calls to malloc/new and tags the returned pointer with a random 16-bit value for ensuring temporal memory safety. Upon deallocation, No-FAT intercepts the calls to free/delete and removes the tag bits

[3] Using power-of-two sized objects can eliminate the need for MAST at the cost of additional memory overhead. This is a common optimization that was used in other systems such as Baggy bounds [1].

before calling the allocator's own free/delete API. When pointers are passed to uninstrumented code, tags are ignored by the hardware to maintain compatibility.

B. Compiler Support

Heap Instrumentation. To guarantee spatial protection, we implement an instrumentation pass at the LLVM IR level that replaces program loads and stores with our new instructions, secure_load and secure_store. To prepare the allocation base address register operand, we use simple function-level analysis to propagate the pointers returned by malloc or new intra-procedurally. To handle out-of-context pointers (e.g., those that are loaded from memory or passed as function arguments), our compiler pass inserts compute_base instructions in the corresponding locations to resolve the allocation base address. Our pass inserts verify_bounds instructions in places where a pointer is stored to memory. This can happen due to (a) casting pointer (ptr) to an integer (i.e., `i = (int)ptr`), (b) storing ptr to memory (i.e., `*r = ptr`), (c) passing ptr to a function (`f(ptr)`), and (d) returning ptr from a function (i.e., `return ptr`).

Source-to-Source Transformation. In order to achieve intra-allocation memory safety, we use a source-to-source transformation (Buf2Ptr), as described in Section II-C. Buf2Ptr is implemented using Clang's rewriter interface. First, we perform an AST traversal over each translation unit to collect a whole program view of composite data types (e.g., structs) and their usages. Then, we perform a second traversal to perform the actual rewriting.

Stack & Global Instrumentation. In order to achieve full memory safety on all memory segments, we extend No-FAT to protect objects that are allocated on the stack and global memory. At compile time, No-FAT instruments all stack and global allocations (e.g., alloca) to use the same bins, which are used to satisfy heap allocations. This way No-FAT uses a unified method to enforce memory safety on all program memory segments. To avoid overheads related to allocating stack objects on the heap, we adopt the same pointer mirroring and memory aliasing techniques used in prior work [19].

C. Operating System Support

MAST Initialization. During program initialization, the memory allocator needs to pass the allocation size information to the hardware. This is a one time task that can be done with a special system call or by writing to a hardware-mapped memory region. The size of the table is fixed, as described in Section IV.

Context Switching. Upon a context switch, No-FAT requires the operating system (OS) to store the MAST (and the dedicated register file contents) of the interrupted process and update the MAST and register file of the new process. Both the MAST and the register file contents are of fixed size and can be stored as part of the process control block. 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).

Privileged Exceptions. When No-FAT's hardware detects an access violation, it throws a privileged exception once the instruction becomes non-speculative. The operating system needs to properly handle this exception as with other privileged exceptions (e.g., page faults). We also assume the faulting address is passed in an existing register so that it can be used for reporting/investigation purposes.

Finally, as No-FAT uses regular data pointers and does not change an object's memory layout, it naturally supports key OS functionalities such as inter-process data sharing, copy-on-write, and memory-mapped files. As No-FAT uses no per-word metadata, it does not require any changes to the page swapping subsystem.

VI. Security Analysis

A. Threat Model

Adversarial Capabilities. We assume a threat model comparable to that used in contemporary related work on memory safety defenses [61], [56], [62], [55], [27]. We assume the victim program to have one or more vulnerabilities that an attacker can exploit to gain arbitrary read and write capabilities in the memory; our goal is to mitigate both spatial and temporal memory violations. Furthermore, we assume that the adversary is aware of No-FAT and has access to the source code, or binary image, of the target program. Finally, we assume that the attacker cannot tamper with the per-process size configurations as they are stored in the MAST and are kept as read-only in kernel memory upon context switch.

Hardening Assumptions. We assume that all hardware components including the ones proposed in this work are trusted and tamper-resistant, and therefore consider attacks that exploit hardware vulnerabilities, such as rowhammer [28] and side-channel attacks [65], to be out of scope. For speculative execution attacks [29], we include Spectre-V1 (aka bounds checking bypass) in our threat model as it violates memory safety (speculatively). We do not include Spectre variants that manipulate branch predictor buffers as No-FAT does not affect program branch behavior.

B. Security Discussion

Buffer Under-/Over-flows. No-FAT defends against the exploitation of buffer overflows (and underflows) by detecting out-of-bounds pointers. No-FAT takes advantage of making the allocation size (per memory page) an architectural feature to enforce spatial memory safety. No-FAT not only protects heap-based allocations, but also stack and global memory regions. To do so, No-FAT reserves alias regions for stack and global objects such that both can use the same allocation size (per memory page) feature. No-FAT's protection applies to both inter- and intra-allocation safety (as Buf2Ptr reduces the intra-allocation problem to inter-allocation).

Use-after-frees. As described in Section II-D, No-FAT provides temporal memory safety by tagging data pointers and validating the tags as part of the spatial bounds checking process. The same allocated virtual/physical memory region can have up to 2^{16} different tags, increasing the chances of catching dangling pointers (as dangling pointers use the old tags of the same allocated region).

Control-Flow Hijacking Attacks. In many attack scenarios, corrupting code pointers becomes a preferred attack vector. For instance, control-flow hijacking attacks, such as ROP [54] and its variants [10], [6], corrupt the return address of a function (or a function pointer) to hijack the control flow of a program. As all of the aforementioned attacks typically start with a spatial/temporal memory safety violation, No-FAT effectively stops control-flow hijacking attacks by eliminating their root cause.

Data-Oriented Attacks. Given a memory safety vulnerability, attackers can launch a data-only attack [11], [24], [46], [12] without abusing any code pointer. No-FAT mitigates those attacks by ensuring that all loads/stores happen between their legitimate bounds. If attackers move a pointer out of bounds to write to a (non-)adjacent allocation, No-FAT throws an exception as the computed base address of the malicious pointer does not match the base address operand of the `secure_load`/`secure_store` instructions.

Uninitialized Reads. No-FAT does not explicitly mitigate uninitialized read attacks, in which attackers can leak information from stack/heap locations by loading from these locations before doing a store operation. To mitigate this attack vector, No-FAT requires that deallocated objects (heap or stack) be zeroed out. Prior work showed that this process can be done efficiently in software [36].

Third-party Library Attacks. While No-FAT maintains full compatibility with third party libraries that are not instrumented with our compiler pass, we offer no security guarantees about vulnerabilities that exist in such uninstrumented code. To increase the security coverage, we create software wrappers for commonly used memory functions that appear in third-party libraries (e.g., `memcpy`, `memset`, and `memove`) to ensure that they cannot be used by an attacker to undermine No-FAT. For example, Listing 3 shows the pseudocode for our `memcpy` wrapper that first computes the base address of the source/destination pointer and ensures it matches the base address of the source/destination plus size before calling the original `memcpy` function.

```
void *memcpy_wrap(void *dst, void *src, size_t n){
  compute_base src, src_base
  compute_base src+n, src_end_base
  Assert(src_base == src_end_base)
  compute_base dst, dst_base
  compute_base dst+n, dst_end_base
  Assert(dst_base == dst_end_base)
  return memcpy(dst, src, n);
}
```

Listing 3: Example `memcpy` wrapper.

C. Spectre-V1 Resiliency

A key advantage of No-FAT over prior memory safety defenses is its natural resiliency to certain classes of speculative side-channel attacks, namely Spectre-V1 (bounds checking bypass) [29]. We first summarize how Spectre-V1 works. Then, we show how it can undermine prior memory safety techniques. Finally, we describe how No-FAT mitigates it with no extra cost.

```
1 if (i < a->length) { // mispredicted branch
2   secret = a->data[i];
3   val = b[64 * secret]; // secret is leaked
4 }
```

Listing 4: Example speculative execution attack.

Attack Summary. To better understand how Spectre-V1 works, let us consider the example shown in Listing 4, in which the attacker controls the index, `i`. The attacker first trains the branch predictor by supplying multiple valid values for `i` (i.e., less than `a->length`). Then, the attacker provides an out-of-bounds index `i > a->length`. While this index violates the software bounds check in Line 1, the hardware will mispredict the condition (i.e., branch is taken) and speculatively executes Lines 2 and 3. As a result, a speculative buffer overread occurs at Line 2 and the read value (`secret`) is used as an index at Line 3. The attacker finally leaks the `secret` value via a covert channel as speculative execution leaves traces in processor structures (e.g, data caches). For example, the address in Line 3 depends on the `secret`, thus flushing and reloading the L1 data cache will allow the attacker to find out which cache line was used and reveals the secret.

Prior Work. Spectre-V1 is a main concern for prior memory safety techniques as it can be used to undermine their security guarantees. For example, attackers can infer the memory tag value of memory without triggering a memory tagging violation and use that to bypass memory tagging solutions (i.e., SPARC ADI [45] and ARM MTE [2]) [5]. To mitigate Spectre-V1, prior work suggested inserting serialization instructions (aka fences) at certain program points to prevent the processor from speculatively bypassing bounds checks [8]. This approach can result in up to 10x runtime overheads [44]. Another line of work proposed isolating speculatively accessed data to prevent leakage via covert-channels [67], [4]. While these defenses reduce the performance overheads, they add substantial complexities to the hardware design.

No-FAT vs. Spectre-V1. No-FAT's `secure_load` and `secure_store` instructions are resilient against Spectre-V1 by construction. Even if the processor mispredicted the branch instruction in Line 1 of Listing 4, the `secure_load` that is used in Line 2 holds the legitimate base and bounds of `a->data` as a third operand. Thus, it immediately recognizes the speculative access as an out-of-bounds access and does not allow `a->data[i]` to access the cache (to avoid modifying the cache state). Hence, No-FAT is resilient against the recent Spectre attack, namely the μop Disclosure Primitive, which exploits the micro-op cache as a timing channel to transmit

TABLE I: Area, delay and power overheads of No-FAT (GE represents gate equivalent).

Hardware Structure	Area (GE)	Delay (*ns*)	Power (*mW*)
Baseline L1 data cache	503,914	1.99	29.7
Bounds checking module	32,130	0.81	1.16
Base computing module	27,346	1.50	1.17

out-of-bounds secrets [48]. Additionally, No-FAT prevents the dependent load instruction from executing by unmarking the ready bit on the register that has the load value (then raising an exception when the out-of-bounds access is non-speculative). We delay raising the exception until the commit stage to avoid false alarms (i.e., if the out-of-bounds memory access happens due to a benign branch misprediction).

Other Spectre Variants. As stated in Section VI-A, No-FAT does not protect against Spectre-variants other than V1 (bounds checking bypass) as the main focus in this work is memory safety. Examples of other Spectre variants include Spectre-V2 (aka branch target injection), which can be used by an attacker to pollute the branch target buffer and force the victim program to speculatively jump to an arbitrary sequence of instructions (called a Spectre gadget). If the Spectre gadget has memory access instructions (e.g., `secure_loads`), they will be speculatively executed based on the current contents of register `R1` (memory address) and register `R3` (allocation base address) even if those register contents belong to an incorrect execution context. The same argument applies if the Spectre gadget includes `compute_base` instructions. Other Spectre-V2 mitigations can be used to address this attack vector [30].

VII. Evaluation

We evaluate No-FAT across multiple dimensions. First, we measure the hardware overheads of No-FAT. Second, we compare the performance of No-FAT against state-of-the-art pre- and post-deployment memory safety solutions using SPEC CPU2017. Third, we analyze No-FAT's memory overheads. Fourth, we evaluate Buf2Ptr by estimating its memory and performance for all benchmarks.

A. Hardware Overheads

No-FAT requires minimal hardware changes. Qualitatively, No-FAT requires a 1KB `MAST` and extra logic to compute the allocation base address (namely, one subtract, one shift, two 64-bit multipliers) and the bounds checking module (namely, one subtract, one shift, one 64-bit comparator, and one 16-bit comparator). As the bounds checking operations happen in parallel to the L1 data and tag accesses, processor clock frequency should not be impacted. We quantified these overheads by adding No-FAT to a typical energy optimized 32KB direct mapped L1 cache. We implement our modules using Verilog and synthesize them with the Synopsys design compiler and the 45nm NangateOpenCell library. We generate the SRAM arrays (for `MAST` and the tag/data arrays) with OpenRAM [23].

Table I summarizes our VLSI implementation results. The timing delay of the bounds checking module is minimal ($0.81ns$) as it uses a pipelined design that first fetches the allocation size from `MAST` and then does a subtraction followed by comparison operations. This latency can be overlapped with the access latency of L1 cache. The bounds checking module adds 6% additional area compared to the L1 data cache. This area is dominated by the SRAMs of `MAST` and the two comparators. On the other hand, the base computing module (which is invoked by the `compute_base` instruction) area is dominated by the 64-bit multiplier. The module latency can be further optimized with a more customized multiplier.

B. Software Performance Overheads

Our VLSI measurements show that No-FAT hardware modifications add no performance overhead. Here, we evaluate the software-based overheads. No-FAT instructions `secure_load` and `secure_store` are similar to regular loads and stores. Thus, they do not increase code size. While our instructions use one more register operand compared to regular memory instructions, the extra register pressure is compensated for by adding a No-FAT-specific register file (i.e., similar to Intel MPX). The additional functionality performed by our instructions can be totally hidden within the processor pipeline as shown in Section VII-A. However, No-FAT requires a binning memory allocator and invokes additional instructions (`verify_bounds` to verify pointer bounds before storing them to memory and `compute_base` to compute the allocation base address of arbitrary pointers when they are loaded from memory).

Without loss of generality, we implement No-FAT on top of a simple binning allocator (Binning-Malloc [18]) that divides the virtual memory into 64 regions, each of size 32GB. Each region is used to satisfy heap-allocation requests of a unique size. Stack and global memory allocations are satisfied using special carved out sections of the same 32GB regions. To estimate the `compute_base` instruction overheads, we implement an IR pass using the LLVM/Clang compiler [32] to instrument the code and insert two `mul` instructions followed by a `store` instead of `compute_base` instructions in the corresponding locations. Similarly, we insert dummy `store` instructions in place of `verify_bounds` instructions. We use a `store` to make sure the instruction is not omitted by compiler optimizations.

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 implement No-FAT using `Clang 4.0.0` and compare it against AddressSanitizer (ASan) and Intel MPX, as representatives of pre- and post-deployment memory safety solutions, respectively. Each tool is run using its best recommended settings [43]. We run each tool such that it suppresses its warnings or errors so that benchmarks run to completion. Additionally, we disable any reporting to minimize the performance impact this functionality may have. Given the difference in compiler versions and optimization levels that each tool supports, we

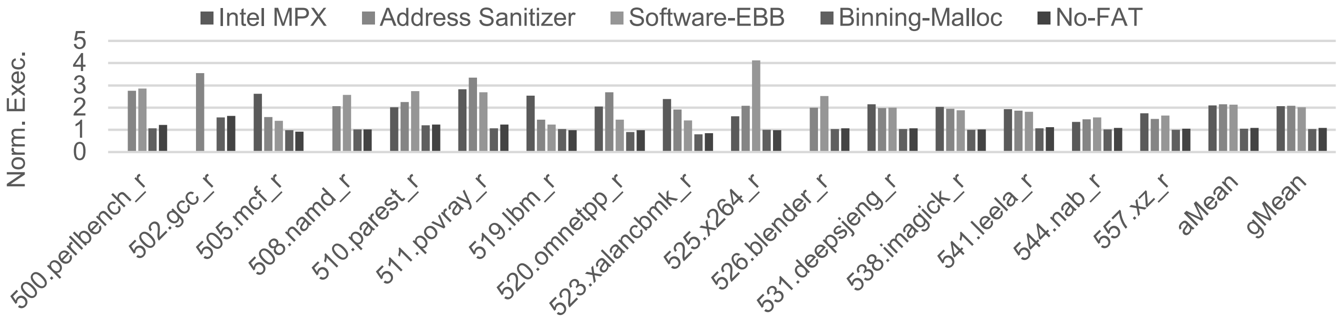


Fig. 3: Performance overheads of the SPEC CPU2017 benchmarks for different tools normalized to their corresponding baseline.

normalize each against their respective baselines for proper comparison.[4] To get better insight on No-FAT overheads, we also run a software-only version of No-FAT that explicitly checks pointer bounds in software with no hardware support (Software-EBB) and a malloc-only version that only uses the binning memory allocator with no bounds checking (Binning-Malloc). We use the SPEC CPU2017 benchmark suite 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.

Performance Results. Figure 3 summarizes the performance overheads of SPEC CPU2017 for different tools normalized to their corresponding baseline. The geometric mean of each tool is as follows: ASan (2.07x), MPX (2.06x)[5], Software-EBB (2.0x), Binning-Malloc (1.04x) and No-FAT (1.08x). The main reason for No-FAT overheads comes from the underlying memory allocator, which introduces 1.04x overheads. For example, `gcc` allocates many small objects that are padded to the nearest Binning-Malloc size. As a result it introduces 62% extra runtime with No-FAT, whereas its Binning-Malloc version has a 55% slowdown. Configuring the allocation sizes at program initialization should reduce the padding and the overheads.

C. *Software Memory Overheads*

To accurately measure the memory usage of No-FAT, we use a Linux-based utility, Syrupy, that regularly takes snapshots of the memory of a running process [57]. We measure the peak resident set size (RSS) to get the actually used memory rather than virtual address space which is reserved. Table II shows that the No-FAT's binning allocator only adds 6.66% memory overheads on average compared to the `stdlib` allocator with `gcc`, `parest`, `povray` as outliers. We inspect those allocation intensive benchmarks by running them with six different memory allocators (including No-FAT). Figure 4 shows that No-FAT memory overheads are comparable to other binning (i.e., Jemalloc [20] and TCmalloc [21]) and non-binning allocators (i.e., Dlmalloc [33] and Scudo [35]).

TABLE II: Memory usage for SPEC CPU2017.

Bench.	**Memory usage (MB)**	**# of Heap allocations**
	No-FAT	Buf2Ptr
perlbench	[+3.27%] 160.80	[+48.7E0] 54.2E6
gcc	[+20.96%] 1,555.57	[+199.3E3] 2.7E6
mcf	[+0.07%] 610.64	[0.0E0] 495.4E3
namd	[-3.71%] 156.53	[0.0E0] 20.2E3
parest	[+26.05%] 527.07	[+107.4E6] 265.1E6
povray	[+35.04%] 8.75	[+10E0] 63.4E3
lbm	[+0.04%] 411.66	[0.0E0] 2.0E0
omnetpp	[+3.79%] 251.36	[+1.7E6] 454E6
xalancbmk	[+6.95%] 512.83	[0.0E0] 138.4E6
x264	[+1.50%] 159.40	[+1.5E0] 2.2E3
blender	[+12.42%] 710.61	[+3.4E6] 9.1E6
deepsjeng	[+0.25%] 702.54	[+15.0E6] 15.0E6
imagick	[-0.91%] 285.05	[+1.0E0] 9.3E6
leela	[-7.01%] 23.56	[+1.2E3] 53.8E6
nab	[+7.74%] 159.03	[+38.1E3] 374.5E3
xz	[+0.04%] 727.30	[+0.0E0] 41.0E0

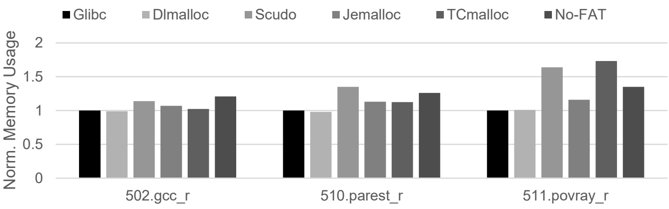


Fig. 4: Memory usage for the three allocation-intensive benchmarks with different memory allocators.

D. *Buf2Ptr Analysis*

Memory. The memory overheads of Buf2Ptr are reported separately. As Buf2Ptr promotes intra-allocation buffers to standalone allocations, it adds additional heap allocations as reported between brackets in the last column of Table II. The majority of benchmarks add few extra allocations with the exception of `parest`, `blender`, and `deepsjeng`. The latter is interesting as it performs a single malloc call to 15 million structs, each with one intra-buffer. So, even though the extra allocations look large, the actual source code transformation is minimal as it only affects one struct.

Performance. The performance overheads of promoting intra-allocation fields to standalone allocations are amortized over program execution. For example, a buffer field

[4] We use `Clang 7.0` for ASan and `GCC 7.3.1` for Intel MPX.

[5] `gcc`, `perlbench`, `namd`, and `blender` failed to run with MPX due to unrecoverable errors. Thus, we exclude them from MPX averages.

of size 64B typically requires eight 8B loads in regular execution. With Buf2Ptr, one can argue that 16 8B loads will be needed as every load now passes through one level of indirection. However, as we implement Buf2Ptr as a source level transformation, we take advantage of the compiler to optimize the access to only 9 8B loads (i.e., one load to get the new base address followed by the original 8 loads with their address adjusted to the new base). We verify this hypothesis by measuring the overhead of implementing Buf2Ptr for the C programs in SPEC CPU2017 benchmarks. The overheads are less than 1% compared to a baseline that uses the same memory allocator (without Buf2Ptr).

VIII. Deployment Considerations

In this section, we discuss the system requirements, strengths, and weaknesses of No-FAT.

System Requirements. No-FAT requires the usage of binning memory allocators. While the internal details of the memory allocator are irrelevant to No-FAT design, the minimum requirement is to have allocations of the same sizes per any memory page. Some allocators (aka non-binning or header-based allocators) violate the above requirement by allocating objects of different sizes in the same page. The non-binning allocators rely on an allocation header to keep track of each allocation size. These allocators are not protected by No-FAT as the cost of deriving allocation base addresses will be much higher (requiring a memory access to the allocation header every time any memory instruction is executed).

Additionally, while porting No-FAT's spatial memory protection to non-64-bit systems is possible, the temporal memory aspect strictly requires 64-bit systems in order to store the temporal tags in the upper bits of the data pointers. On non-64-bit systems, temporal memory safety can be achieved with less efficient approaches such as free list randomization and memory quarantining.

Handling Gnarly, Gory C Idioms. Some C programmers have the propensity to exploit undefined behaviors. By undefined behaviors we mean behaviors that are not explicitly disallowed in the standard. These issues are documented in an excellent exposition by the authors of the CHERI system [13]. One of the most common of these idioms is the case of intentionally creating out-of-bounds pointers. Although it is unclear why programmers follow this idiom, it exists, and we strive to identify such cases.

Consider the following: `q = p + 100; x = Bar(q);`. Here, `p` is a pointer to a 32B allocation. After the arithmetic operation, `q` is an out-of-bounds pointer that will be passed to function `Bar`. Inside `Bar`, the program may do `z = q - 100` before using `z` to access memory. Since No-FAT only uses intra-procedural analysis, `Bar` will recompute the base address for `q` at function entry using `compute_base`. However, as `q` is already an out-of-bounds pointer, the resultant base address will be wrong (i.e., it will not point to the original object pointed to by `p`). We refer to this case as a data pointer escape operation. If the compiler performs only intra-procedural analysis to determine and encode pointers into checking loads and stores, it will result in an insufficient check. To handle this case, No-FAT uses `verify_bounds` to catch out-of-bounds pointers before they escape to memory (or a different function), as described in Section II-E. This functionality should help programmers catch undefined behavior and fix it.

TABLE III: Categorization of prior work on spatial memory safety based on how they handle the security metadata.

	Base & Bounds		**Tripwires**
	Per-Allocation	**Per-Pointer**	
Disjoint Metadata	Baggy Bounds [1]	Hardbound [14] Softbound [41] Watchdog [39] Intel MPX [43] AOS [27] CHEx86 [55]	ASan [52]
Inlined Metadata	EffectiveSan [17] In-Fat [64]	CHERI [61], [62] Compact-Ptrs [31]	REST [56] Califorms [50]
Co-joined Metadata	ARM MTE [2] SPARC ADI [45]		
Implicit Metadata	Native-Ptrs [16], [19]	**No-FAT**	

Strengths. Compared to many other systems, No-FAT provides deterministic memory safety guarantees at the finest granularity. No-FAT provides protection against a wide variety of spatial/temporal memory safety violations including control flow hijacking attacks, data oriented attacks, and pure data corruption. No-FAT's protection comes with minimal performance overheads and minor hardware changes. Furthermore, No-FAT naturally mitigates a common speculative execution threat (Spectre-V1) at no additional cost.

Weaknesses. Buf2Ptr requires the precise type information of an allocated object. While this is guaranteed for C++ objects, it is not always possible in C-style programs where `void*` allocations may be used. In these cases, the compiler may not be able to infer the correct type, in which case intra-allocation support may be skipped. This is a common limitation for techniques that rely on source-level transformations for intra-allocation protection [50]. Our evaluation results in Section VII-D show that the number of intra-allocation buffers is minimal compared to total allocations. Cases with ambiguous types are not common and can be properly handled with program annotations, if needed. We leave this part to future work.

IX. Comparison with Prior Work

In this section, we summarize prior work related to memory safety mitigations and show how No-FAT is different. We first categorize prior work based on how they handle the metadata (e.g., base and bounds information), as shown in Table III. The metadata can be maintained in one of the following forms: disjoint, inlined, co-joined, or implicit. We then compare No-FAT with prior techniques in terms of security, hardware

TABLE IV: Comparison with prior works.

Proposal	Spatial Protection *		Temp.	Binary	MT	Hardware	Metadata	Memory	Performance
	Inter	Intra	Prot. §	Comp. †	Support ¶	Modifications	Overhead	Overhead	Overhead
Hardbound [14]	●	● ‡	○	○	◒	μop injection, L1$ & TLB for tags	0-2 words per ptr & 4 bits per word	∝ # of ptrs	∝ # of ptr derefs
Softbound [41]	●	◒ ‡	○	○	◒	N/A	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
Baggy Bounds [1]	●	○	○	●	●	N/A	N/A.	∝ padding objects to the nearest size	∝ # of ptr ops
Compact-Ptrs [31]	●	○	○	○	●	One extra pipeline stage for bounds check & update	N/A	∝ padding objects to the nearest size	∝ # of ptr ops
Watchdog [39]	●	◒ ‡	●	○	◒	Renaming logic, μop injection logic and Lock location$	4 words per ptr	∝ # of ptrs and allocs	∝ # of ptr derefs
WatchdogLite [40]	●	◒ ‡	●	○	◒	N/A	4 words per ptr	∝ # of ptrs and allocs	∝ # of ptr ops
Native-Ptrs [16], [19]	●	○	○	●	●	N/A	N/A.	∝ padding objects to the nearest size	∝ # of ptr ops
In-Fat [64]	●	●	○	●	●	32 96-bits bounds registers, a new execution unit, and 10 instructions	16B per object	∝ # of objects	∝ # of ptr ops and # of ptr derefs
Intel MPX [43]	●	◒ ‡	○	◒	◒	Unknown (closed platform)	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
BOGO [68]	●	◒ ‡	●	◒	◒	Unknown (closed platform)	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
CHERI [61], [62]	●	◒ ‡	○	○	●	Capability coprocessor, Tag$ and Capability Unit	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
CHERIvoke [63]	○	○	●	○	●	Capability coprocessor, Tag$ Tag controller, and Capability Unit	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
PUMP [15]	●	○	●	◒	●	Extend all data units by tag width, new miss handler and Rule$	8B per cache line	∝ prog. mem. footprint	∝ # of ptr ops
ARM MTE [2]	●	○	◒	●	●	Unknown (closed platform)	4 bits per 16B objects	∝ prog. mem. footprint	∝ # of tag (un)set ops
REST [56]	◒	○	◒	●	●	1-8B per L1D line, 1 comparator	8–64B token	∝ blacklisted memory	∝ # of (dis)arm insns.
Califorms [50]	◒	◒	◒	●	●	8B per L1D line, 1 bit per L2/L3 line	1-7B per critical field	∝ blacklisted memory	∝ # of BLOC insns.
AOS [27]	●	○	◒	◒	◒	ARM PAC instructions, memory check queue, bounds$, and bounds way buffer	8B bounds per ptr	∝ # of ptrs	∝ # of ptr derefs
CHEx86 [55]	●	◒	◒	●	◒	μop injection logic, Alias$, Capability$, and Speculative Pointer Tracker	2 words per ptr	∝ # of allocs & ptrs	∝ # of ptr derefs
No-FAT	●	●	◒	◒	●	bounds checking & base computing modules and base address register file	1KB per process Table	∝ padding objects to the nearest size	∝ # of ptr derefs

[*] ● - Complete (Linear and non-linear overflows); ◒ - Linear only; ○ - No protection.
[§] ● - Complete; ◒ - Partial protection; ○ - No protection.
[†] ● - Fully compatible; ◒ - Execution compatible, but protection dropped when external modules modify pointer; ○ - No support.
[¶] ● - Supported (stateless); ◒ - Supported (requires synchronization on global metadata); ○ - No support.
[‡] Achieved with bounds narrowing.

complexity, memory requirements, and performance overheads in Table IV.

Explicit Base & Bounds. This class of memory safety defenses attaches bounds metadata to every pointer or allocation. The metadata can be stored in a shadow (aka disjoint) memory region (e.g., Hardbound [14], Softbound [41], Intel MPX [43], CHEx86 [55], and AOS [27]) or be marshaled with the pointer by extending its size (e.g., CHERI [61]). Temporal memory safety can be added to the above techniques by either storing an additional "identifier" along with the pointer metadata and verifying that no stale identifiers are ever retrieved (e.g., CETS [42], Watchdog [39], and WatchdogLite [40]) or invalidating all pointers to freed regions in the lookup tables (e.g., BOGO [68]).

While explicit base and bounds systems offer similar security guarantees to No-FAT, our solution has the advantage of requiring simpler hardware modifications and being faster than prior systems [43], [62], [55], [27]. The above savings mainly stem from the fact that No-FAT uses no metadata for spatial/temporal memory safety. Disjointly storing the metadata in a shadow memory [14], [43], [27], [55] requires extra memory accesses to fetch and update the metadata and introduces atomicity problems for multithreading applications.

Other base and bounds techniques, such as Softbound [41], can take advantage of hardware support (similar to No-FAT) by (1) encoding the pointer base and bounds in extra `load`/`store` register operands, (2) propagating them within function scope, and (3) performing the memory safety checks in hardware. Such hardware support can reduce the runtime overheads of Softbound by eliminating the need for performing the bounds checking in software and reducing the number of memory lookups to fetch the metadata from the disjoint memory structures. However, even with hardware support the aforementioned solution needs to perform two extra memory operations (to access the base and bounds) every time a pointer is loaded from memory (i.e., not within the same function scope in which the pointer was created). In contrast, No-FAT has high performance due to how it derives the base and bounds information from the pointer itself when it is loaded from memory. Specifically, No-FAT replaces the costly metadata memory accesses with simple arithmetic computations by using the `compute_base` instruction.

On the other hand, increasing the pointer width to include the metadata [61], [62] changes object layouts and breaks compatibility with the rest of the system (e.g., unprotected libraries). On the contrary, No-FAT performs simple arithmetic computations to derive the allocation bounds and uses a fixed area cost for `MAST`. Furthermore, the metadata-less aspect of our scheme allows us to support multi-threading applications with no false positives/negatives, which occur in disjoint metadata schemes (e.g., Intel MPX [43]). Additionally, our Buf2Ptr transformation implictly resolves the intra-allocation memory safety problem, which is overlooked by recent memory safety techniques [55], [27].

Software-based Implicit Base & Bounds. The idea of deriving allocation bounds from the pointer itself is not new. For example, guarded pointers divided memory into powers-of-two segments and encoded the segment size into the pointer's upper bits [9]. Similarly, baggy bounds [1] restricts allocation sizes to powers-of-two and encodes the binary logarithm of the allocation size in the pointer's upper bits. Unlike No-FAT, this design choice significantly increases the program's memory footprint due to padding allocations to the nearest powers-of-two size. Moreover, neither guarded pointers nor baggy bounds offers temporal protection.

Compact-pointers [31] tried to avoid the powers-of-two restriction by using a floating-point representation to encode allocation bounds in the pointer itself. CHERI-concentrate [62] adopts a similar approach to compress metadata to 128 bits (instead of 256 bits) by changing a pointer's layout and introducing instructions to manipulate them. Due to the pointer layout manipulation, both techniques neither support temporal memory safety nor maintain binary compatibility.

Similar to No-FAT's binning allocator, Native-Pointers [16], [18], [19] divides the program's virtual address space into several regions of equal size and uses each region to allocate objects of similar non powers-of-two sizes. As a software-only solution, Native-Pointers suffers from high performance overheads. Additionally, Native-Pointers does not naturally provide temporal protection. A follow-up work (EffectiveSan [17]) adds temporal protection (and intra-allocation memory safety) to Native-Pointers but with expensive per-allocation metadata. Concurrent to our work, Xu et al. add hardware support for EffectiveSan, dubbed In-Fat [64]. The key idea is to maintain a per-allocation metadata table and use the pointer's upper bits to index into this table for intra-allocation bounds retrieval. In-Fat uses different metadata schemes for different program objects (e.g., stack, heap, and globals) to reduce the lookup overhead. Unlike No-FAT, In-Fat does not provide temporal protection as it utilizes the pointer's upper bits for indexing into the metadata tables. A key advantage of No-FAT over EffectiveSan and In-Fat is that it does not require any per pointer/allocation metadata. Thus, it runs with almost native performance, making it best suited to be an always-on memory safety mitigation.

Memory Tagging. This class of techniques associates a "color" with newly allocated memory, and stores the same color in the upper bits of the data pointer that is used to access the allocated memory. At runtime, the hardware enforces spatial memory safety by comparing the colors of the pointer and accessed memory. For example, SPARC ADI [45] assigns 4-bit colors to every 64B of memory (i.e., limiting the minimum allocation size to 64B), while ARM MTE [2] uses 8-bit colors per every 16B of memory [53]. Since metadata bits are acquired along with the corresponding data, no extra memory operations are needed.

Temporal safety is enforced by assigning a different color when memory regions are reused. Unlike No-FAT, which uses a 16-bit tag for temporal protection, the number of tag bits in memory tagging defenses is limited as the tags are used for pointers and memory locations. As a result, prior techniques offer less entropy for temporal protection. For example, in ARM MTE colors will be repeated every 255 allocations, while in SPARC ADI colors are repeated every 15 allocations, raising the attacker's chances of bypassing the defense.

Tripwires. This class of memory safety defenses aims to detect overflows by marking the memory regions on either side of an allocation, and flagging accesses to them. For example, REST [56] stores a predetermined 8–64B random number, dubbed a token, in the memory to be invalidated. Spatial memory safety violations are detected by comparing cache lines with the token when they are fetched. Due to its large token size, REST does not support intra-allocation memory safety. Califorms [50] solves this problem by inserting 1–7B hardware canaries between object fields and introducing cache line formats to inline metadata within the program data itself.

While state-of-the-art tripwires systems (i.e., REST [56] and Califorms [50]) come with comparable performance overheads to No-FAT, our solution offers better security guarantees as it cannot be bypassed by non-adjacent buffer overflows, which represent 27% of Microsoft's memory safety CVEs [5].

Additionally, REST and Califorms demand a quarantine pool to provide temporal safety. Using memory quarantining typically increases performance overheads as it prevents the program from reusing recently freed memory to satisfy new allocation requests. No-FAT instead does not rely on memory quarantining to achieve temporal memory safety, effectively reducing the overheads for allocation intensive applications. Finally, attackers can bypass Califorms' intra-allocation protection if the binary is leaked as the locations of the hardware canaries are encoded in the binary itself, whereas No-FAT does not keep the binary secret as allocation information is derived and propagated at runtime.

X. Conclusion

In this paper we proposed No-FAT, a secure architecture for implicitly deriving allocation bounds. No-FAT enforces spatial memory safety and a degree of temporal safety without increasing program memory footprint, while maintaining full compatibility with unprotected code. Overall, No-FAT incurs 8% performance degradation compared to a 100% slowdown for its software version, while providing extra security guarantees. This has the benefits of reducing fuzz-testing overheads to improve pre-deployment software testing. Furthermore, if end users are willing to pay 8% performance degradation for memory safety protection, then No-FAT is an excellent solution. On the other hand, if users are willing to trade off some security features (e.g., non-pointer data corruption) for no performance degradation, lightweight exploit mitigation techniques, such as ZeRØ [58], can be used. The benefits of No-FAT go well beyond memory safety: for instance, having the allocation size as an architectural feature can help accelerate garbage collectors for memory safe languages; it also provides an opportunity for enhancing the predictability of memory prefetchers and DRAM controllers.

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] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[2] 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, [Online; accessed 01-May-2021].

[3] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, Orlando, FL, USA, 1994, pp. 290–301.

[4] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Isolating speculative data to prevent transient execution attacks," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 178–181, 2019.

[5] 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

[6] 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, 2011, pp. 30–40.

[7] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 713–724.

[8] C. Carruth, "RFC: Speculative load hardening (a spectre variant 1 mitigation)," 2018. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html

[9] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *SIGPLAN Not.*, vol. 29, no. 11, pp. 319–327, Nov. 1994.

[10] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, September 2010, pp. 559–572.

[11] 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.

[12] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao, "Exploitation techniques and defenses for data-oriented attacks," in *Proceedings of the 2019 IEEE Cybersecurity Development (SecDev)*, Tysons Corner, VA, USA, September 2019, pp. 114–128.

[13] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe c abstract machine," in *ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Istanbul, Turkey, 2015, pp. 117–130.

[14] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, "Hard-Bound: 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.

[15] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *ASPLOS '15: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[16] G. J. Duck and R. H. C. Yap, "Heap bounds protection with low fat pointers," in *CC '16: Proceedings of the 25th International Conference on Compiler Construction*, Barcelona, Spain, 2016, pp. 132–142.

[17] G. J. Duck and R. H. C. Yap, "EffectiveSan: type and memory error detection using dynamically typed C/C++," in *PLDI '18: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018.

[18] G. J. Duck and R. H. C. Yap, "An extended low fat allocator API and applications," *arXiv preprint arXiv:1804.04812*, 2018.

[19] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers," in *NDSS '17: Proceedings of the 24th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2017.

[20] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *Proceedings of the Technical BSD Conferene*, 2006. [Online]. Available: https://www.bsdcan.org/2006/papers/jemalloc.pdf

[21] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc," 2007. [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[22] GoogleProjectZero, "0day in the wild," 2019. [Online]. Available: https://googleprojectzero.blogspot.com/p/0day.html

[23] 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.

[24] 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.

[25] R. Hundt, S. Mannarswamy, and D. Chakrabarti, "Practical structure layout optimization and advice," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, USA, 2006, pp. 233–244.

[26] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," in *ASPLOS XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, 2006, pp. 207–218.

[27] 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.

[28] 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.

[29] 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.

[30] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "SpecCFI: Mitigating spectre attacks using CFI informed speculation," in *SP '20: Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2020, pp. 39–53.

[31] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *CCS '13: Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[32] 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.

[33] D. Lea, "A memory allocator," 2000. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[34] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," Microsoft, Tech. Rep. MSR-TR-2019-18, June 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/

[35] LLVM, "Scudo hardened allocator." [Online]. Available: https://llvm.org/docs/ScudoHardenedAllocator.html

[36] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities," in *NDSS '17:*

Proceedings of the 24th Annual Network and Distributed System Security Symposium, February 2017.

[37] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," 2019. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[38] MozillaSecurity, "VIRGO: Crowdsourced fuzzing cluster," 2019. [Online]. Available: https://github.com/MozillaSecurity/virgo

[39] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdog: hardware for safe and secure manual memory management and full memory safety," in *ISCA '12: Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[40] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "WatchdogLite: hardware-accelerated compiler-based pointer checking," in *CGO '14: Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[41] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *PLDI '09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009, pp. 245–258.

[42] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler enforced temporal safety for C," in *ISMM '10: Proceedings of the 2010 International Symposium on Memory Management*, Toronto, Ontario, Canada, 2010, pp. 31–40.

[43] 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.

[44] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You shall not bypass: Employing data dependencies to prevent bounds check bypass," *arXiv preprint arXiv:1805.08506*, 2018.

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

[46] 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.

[47] F. Qin, S. Lu, and Y. Zhou, "SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs," in *HPCA '05: Proceedings of the IEEE 11th International Symposium on High Performance Computer Architecture*, 2005.

[48] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via Intel/AMD micro-op caches," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, June 2021.

[49] P. Roy and X. Liu, "StructSlim: A lightweight profiler to guide structure splitting," in *CGO '16: Proceedings of the 2016 International Symposium on Code Generation and Optimization*, Barcelona, Spain, March 2016, p. 36–46.

[50] 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, p. 558–571.

[51] D. L. Schacter and E. Tulving, *Memory Systems 1994*. MIT Press, 1994.

[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker," in *ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[53] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *arXiv preprint arXiv:1802.09517v1*, February 2018.

[54] 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, Virginia, USA, 2007, pp. 552–561.

[55] 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.

[56] 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.

[57] J. Sukumaran, "Syrupy," 2008. [Online]. Available: https://github.com/jeetsukumaran/Syrupy

[58] M. Tarek Ibn Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZeRØ: Zero-overhead resilient operation under pointer integrity attacks," in *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*, Worldwide Event, June 2021.

[59] V. Tsyrklevich, "GWP-ASan: Sampling heap memory error detection in-the-wild." [Online]. Available: https://www.chromium.org/Home/chromium-security/articles/gwp-asan

[60] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory," in *ISCA '18: Proceedings of the 45th Annual International Symposium on Computer Architecture*, Los Angeles, CA, USA, 2018, pp. 207–220.

[61] 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.

[62] 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.

[63] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and et al., "CHERIvoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, October 2019, pp. 545–557.

[64] S. Xu, W. Huang, and D. Lie, "In-Fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection," in *ASPLOS '21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual Event, 2021.

[65] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *SEC '14: Proceedings of the 23rd USENIX Conference on Security Symposium*, San Diego, CA, USA, 2014, pp. 719–732.

[66] C. Yu, P. Roy, Y. Bai, H. Yang, and X. Liu, "LWPTool: A lightweight profiler to guide data layout optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2489–2502, November 2018.

[67] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data," in *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, Columbus, OH, USA, October 2019, pp. 954–968.

[68] T. Zhang, D. Lee, and C. Jung, "BOGO: Buy spatial memory safety, get temporal memory safety (almost) free," in *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, USA, 2019, p. 631–644.