BSan: A Powerful Identifier-Based Hardware-Independent Memory Error Detector for COTS Binaries

Wen Zhang
University of Georgia

Botang Xiao
University of Georgia

Qingchen Kong
University of Georgia

Le Guan
University of Georgia

Wenwen Wang
University of Georgia

***Abstract*—This paper presents BSan, a practical software-only memory error detector for binary code. Different from state-of-the-art binary-level detectors, which rely on either the shadow-memory-based approach or the hardware-specific feature and thus suffer from several fundamental limitations, BSan adopts an *identifier-based* approach, enabling it to detect deep memory errors missed by existing detectors. Also, BSan does not depend on any specific hardware features. To reduce the high performance overhead caused by identifier propagation, BSan creates a novel *hybrid* approach, static analysis+dynamic instrumentation, to improve the performance without inheriting the poor reliability of static binary rewriting, distinguishing it from existing detectors that simply refer to static binary rewriting for better performance. The comprehensive evaluation demonstrates that BSan can detect more memory errors than state-of-the-art binary-level detectors. Meanwhile, the performance and memory overheads of BSan are comparable to those of existing detectors.**

I. Introduction

Memory errors, such as buffer overflows and use-after-free errors, have serious impacts on software reliability, stability, and security [2], [4], [36], [37], [39]. A previous study shows that around 70% of high-severity security bugs found in Google Chrome, one of the most popular web browsers, are memory errors [29]. This phenomenon is also observed by a security vulnerability study conducted by Microsoft [23]. Therefore, detecting memory errors is of high importance for building reliable, secure, and sustainable software systems.

On today's commercial platforms, such as Microsoft Windows and Apple MacOS, most software is still closed-source and distributed only through executable binaries. Even on open-source platforms like Linux, many applications depend on third-party libraries without source code available. Further, legacy software developed for outdated platforms often lacks complete source code copies. In most cases, although the source code is available, compiling it with modern compiler toolchains may face various problems [20], [35]. Hence, it is strongly necessary to improve the reliability and security of binaries by detecting memory errors lurking in them.

Previous research work has developed several binary-level memory error detectors without the need for application source code. Notable examples include RetroWrite [15], MTSan [10], Dr. Memory [8], Memcheck [32], QASan [18], and Undangle [9]. Although these detectors have been quite successful in detecting memory errors in commercial off-the-shelf (COTS) binaries, they suffer from three fundamental limitations.

Limitation 1: the incapability of detecting deeply hidden memory errors. Most existing binary-level detectors, including QASan, RetroWrite, Dr. Memory, and Memcheck, employ a *shadow-memory-based* approach to capture the status of application memory for memory error detection. Specifically, before a memory access is executed, the corresponding shadow memory location is checked to determine whether the access is valid or not. Although this mechanism is quite intuitive to implement for binary code, it may *miss* both spatial and temporal memory errors due to its limited detection capability (See §II for more details). As a result, deeply hidden memory errors can easily escape these detectors, leading to security risks.

Limitation 2: the lack of universal applicability and practicability. To reduce the high runtime overhead incurred by dynamic binary instrumentation, several detectors, including RetroWrite and MTSan, utilize *static binary rewriting* to instrument target binary code. Although static rewriting can remove the need for dynamic instrumentation, it renders the detectors vulnerable to low practicability, due to the notorious challenges of statically rewriting binaries caused by the inherent complexity of binary code [22], [28]. For instance, RetroWrite only supports position-independent binary code compiled with a specific compiler in a specific version.

Limitation 3: hardware dependence. Different from other software-only detectors, MTSan relies on the memory tagging hardware extension [1], introduced recently by ARMv8.5-A processors, to perform memory error detection. Obviously, this hinders the adoption of this detector on other commercial hardware platforms, e.g., x86 and RISC-V, as well as plenty of ARM devices without such a hardware feature available.

To address the above limitations, this paper presents **BSan**, an *identifier-based* memory error detector for COTS binaries. BSan assigns a globally unique identifier for each memory object and attaches the identifier to pointers that point to this object. When a pointer is dereferenced, BSan first uses the attached identifier to find the metadata of the accessed object, e.g., bounds information, and then examines the metadata and the pointer to detect memory errors. Although prior work has implemented the identifier-based approach at the source code level [24], [25], the development of BSan entails multiple *unique* technical challenges, e.g., how to propagate identifiers along with binary-level pointer operations and how to manage the high performance overhead introduced by the propagation. We will discuss these challenges and our solutions in §III.

Compared to state-of-the-art binary-level detectors, BSan has several distinctive advantages. First, BSan is able to detect *more* memory errors, especially those that are deeply hidden in binaries and cannot be detected using the shadow-memory-based approach. Second, BSan is a pure software detector and does *not* rely on any hardware-specific features. Therefore, it can be applied to binaries compiled for various hardware platforms. Finally, BSan achieves universal practicability by inventing a novel *hybrid* approach, i.e., static analysis+dynamic instrumentation, to avoid brittle static binary rewriting. This also offers an exclusive opportunity for BSan to realize slightly better runtime performance than existing dynamic instrumentation-based detectors, e.g., Dr. Memory.

We have implemented BSan based on two widely used binary analysis and instrumentation frameworks: Dyninst [5] for offline static analysis and DynamoRIO [7] for online memory error detection. To evaluate BSan, we conduct comprehensive experiments, covering three well-recognized benchmark suites, i.e., Juliet [19], SPEC CPU 2017 [12], and PARSEC [6], 22 real-world memory errors, as well as 10 real-world applications in various domains. We also compare BSan with four state-of-the-art binary-level detectors, including MTSan, RetroWrite, Dr. Memory, and Memcheck. Experimental results demonstrate that BSan can achieve desirable detection results by reporting more memory errors than other detectors. Moreover, the detection efficiency of BSan, including both performance and memory overheads, is comparable to existing dynamic instrumentation-based detectors.

In summary, this paper makes the following contributions.

- We present BSan, a powerful software-only memory error detector for COTS binaries. To the best of our knowledge, BSan is the very first binary-level detector that adopts an identifier-based detection approach.
- We overcome multiple unique technical challenges to realize an effective and efficient design of BSan. These challenges differentiate BSan from previous identifier-based detectors developed at the source code level.
- We implement BSan using two popular frameworks Dyninst and DynamoRIO to support x86-64 binaries. We make the source code of our implementation publicly available[1] to facilitate future research on BSan.
- We conduct comprehensive experiments to evaluate BSan. Experimental results show that BSan achieves desirable detection results with both performance and memory overheads in parallel with existing detectors.

II. Background and Motivation

This section first introduces memory errors and then describes the shadow-memory-based and identifier-based memory error detection approaches to motivate the need for a more effective and powerful binary-level detector.

A. Memory Errors

In general, there are two types of memory errors. A *spatial* memory error happens when an instruction attempts to access an address that is out of the bounds of the accessed memory object. Common spatial memory errors include buffer overflows and underflows. Depending on where it is located, the accessed address may or may not be accessible. In the case that the accessed address is accessible, e.g., located in another object, the spatial memory error may *not* trigger any observable misbehavior like a crash. As a result, such memory errors are hard to find if they cannot be caught by a detector.

Differently, a *temporal* memory error occurs when a memory access instruction tries to access a memory object that is out of its liveness range. Use-after-free and double-free errors are two representative temporal errors. Here, it is worth pointing out that use-after-free errors can happen on both heap and stack objects. For instance, a pointer that points to a local variable in a function may be propagated to the outside of the function and dereferenced after the function returns, leading to a stack use-after-free error, also known as use-after-return.

B. Shadow-Memory-Based Detection

The shadow-memory-based detection approach is extensively used in state-of-the-art memory error detectors, including binary-level detectors, such as Dr. Memory [8], Memcheck [32], RetroWrite [15], and QASan [18], as well as source-level detectors, e.g., AddressSanitizer [31].

In this approach, a shadow memory region is allocated to store the *metadata* of the original application memory. The metadata captures the status of the application memory, e.g., whether a specific memory location is allocated and valid for access. The mapping between the application memory and the shadow memory is usually fixed and one-to-one, e.g., a single byte of the application memory is mapped to several bits of the shadow memory, to limit the total size of the shadow memory and make access to the shadow memory simple and fast. The shadow memory is updated at special events to reflect the latest status of the application memory. For instance, when a heap object is deallocated, the shadow memory corresponding to the object is updated to indicate the memory region of the object is freed and thus not valid for access. By checking the metadata stored in the shadow memory when the application memory is accessed, this approach can detect memory errors.

Issues. Although the idea of this shadow-memory-based approach is quite intuitive, it has two critical issues. First, it may miss spatial memory errors. To understand the reason, let us consider the following code snippet:

```c
void foo(int idx, ...) {
  int buf[10];
  buf[idx] = ...; // a buffer overflow if idx >= 10
}
```

In this example, the function argument `idx` is *not* checked before it is used to access the stack buffer `buf`. As a result, this access may lead to a spatial memory error. However, depending on the value of `idx`, the accessed memory location, i.e., `buf[idx]`, may be in the stack frame of a caller function of `foo()`. In that case, checking the shadow metadata of `buf[idx]` *cannot* detect this memory error, as the metadata indicates the address is valid for access. The underlying reason for this problem is that checking the metadata stored in the shadow memory is *insufficient* to validate a memory access.

[1] https://github.com/dvaave/BSan

Second, it may also miss temporal memory errors. As discussed before, a use-after-free error caused by the dereference of a pointer pointing to a deallocated memory object can be detected by checking the shadow metadata. However, if the memory region of the deallocated object is *reused* to allocate a new object, as shown in the following code snippet, simply checking the shadow metadata will not be able to detect the memory error, because the corresponding shadow metadata is about the new object, rather than the old object.

```
p = malloc(sizeof(int));
...
free(p);
q = malloc(sizeof(int)); // reusing the memory of p
*p = ...; // a use-after-free error
```

To mitigate this problem, existing binary-level detectors often adopt a heuristic *quarantine* mechanism [8], which places deallocated objects in a quarantine queue to delay the reuse of the memory. However, even with this mitigation, a use-after-free error can still easily escape the detectors, as memory reuse can happen when the quarantine queue is full. Our experience with existing detectors shows that this is quite common in reality, especially when target applications frequently allocate and deallocate memory objects. Worse, it is very challenging to apply the quarantine mechanism for stack objects at the binary code level, as stack regions are naturally reused across function calls, leading to missed stack temporal errors.

C. Identifier-Based Detection

Different from the shadow-memory-based approach, the identifier-based approach assigns a globally unique *identifier*, e.g., an integer, to each memory object and attaches the identifier to every pointer that points to this memory object. When a pointer is dereferenced, its attached identifier is used to find the metadata of the accessed memory object, which includes the bounds information and whether the object is still alive, i.e., not deallocated. By examining the metadata, this approach can detect both spatial and temporal errors.

The key feature that distinguishes the identifier-based approach from the shadow-memory-based approach is that it maintains the metadata for each memory object and uses the object identifier rather than the object address to find the metadata. This allows it to obtain more accurate metadata, even if the same memory region may be reused for different memory objects. Because of this reason, the identifier-based approach can detect deep memory errors missed by the shadow memory-based approach, e.g., the two memory errors discussed before. Due to the more powerful detection capability, the identifier-based approach has been implemented at the source code level, e.g., SoftBound [24] and CETS [25]. However, as we will discuss in the next section, developing a binary-level identifier-based detector faces unique technical challenges.

III. Technical Challenges

There are three unique technical challenges to developing an identifier-based memory error detector for binary code.

How to attach object identifiers to binary pointers? An important requirement of the identifier-based approach is that every pointer needs to be attached with the identifier of the memory object it points to. At the source code level, this can be easily achieved, e.g., through fat pointers [21], [26], [38]. However, for binary code, it is very challenging, as data layouts of binaries are hard to modify flexibly and arbitrarily, e.g., appending extra bits to every pointer. Considering that 64-bit memory pointers only use the low 48 bits [16], [17], one may think about placing object identifiers into the unused 16 bits of pointers [11]. Unfortunately, this method cannot work either, as it is challenging to differentiate augmented 64-bit pointer values from regular 64-bit non-pointer values at the binary code level, due to the unavailability of the *data type* information in binaries. As a result, it is hard to determine whether the high 16 bits of a 64-bit value should be removed or not before it can be used for an arithmetic/logic operation.

To overcome this challenge, our key observation is that pointers in binary code are stored in either *registers* or *memory locations*. Therefore, by attaching object identifiers to them, we can effectively maintain the association between pointers and object identifiers. Note that we need to use *disjoint* data structures to achieve this. See §IV for more details.

How to propagate identifiers along with binary-level pointer operations? The identifier attached to a pointer needs to be propagated appropriately when the pointer is involved in pointer-related operations. For example, when deriving a new pointer by adding a constant value to an old pointer, the identifier attached to the old pointer needs to be propagated to the new pointer. Otherwise, dereferences of the new pointer cannot be checked. However, identifier propagation at the binary code level is quite challenging, as it is unclear how to propagate identifiers for different instructions. Also, any *incomplete* propagation can potentially cause ineffective detection, resulting in missed memory errors.

To overcome this challenge, we create comprehensive *identifier propagation rules* according to general instruction semantics. Through these propagation rules, we can effectively propagate identifiers along with pointer operations. Note that the design of the propagation rules is *not* tied to any specific instruction sets. See §IV for more details.

How to reduce the heavy performance overhead caused by propagating identifiers? Due to the lack of the data type information in binaries, we have to instrument *all* instructions that may produce new pointers to realize complete identifier propagation. For instance, let us consider the x86-64 instruction `mov %rax,%rbx`, which copies the value in `rax` to `rbx`. Since it is hard to know whether or not the value is a pointer, this instruction has to be instrumented for potential identifier propagation, even if the value may not be a pointer. Note that this is not an issue at the source code level because, with the data type information, this instruction would not be instrumented if the value is not a pointer.

To overcome this challenge, we propose innovative optimizations to reduce the performance overhead introduced by identifier propagation. Our key insight is that it is only necessary to propagate an identifier if the propagated identifier

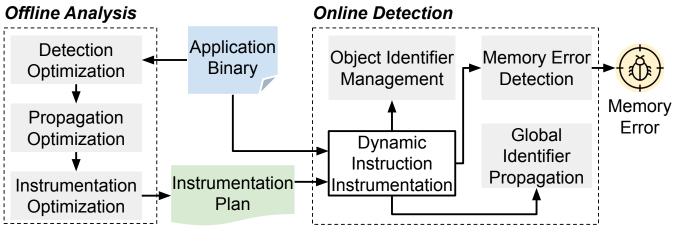


Fig. 1. System overview of BSan.

is required to check memory accesses in the following code for memory error detection. By identifying and removing unnecessary propagation, our optimizations can effectively reduce the performance overhead of propagating identifiers and boost detection efficiency. See §IV for more details.

IV. System Design of BSan

Figure 1 shows the overview of BSan, which consists of two components: *offline analysis* and *online detection*. Given an application binary, BSan first conducts a series of static analyses to identify optimization opportunities, e.g., redundant memory error detection and unnecessary identifier propagation. After the analyses, BSan generates an *instrumentation plan* to note what work needs to be done at specific instructions, e.g., memory error detection and/or identifier propagation. Under the direction of this instrumentation plan, BSan next runs the binary code with dynamic instruction instrumentation to realize object identifier management, global identifier propagation, and memory error detection. If a memory error is detected during the execution, BSan reports it with detailed information, including the error type, the problematic memory access instruction, and the accessed memory object, to facilitate further manual analysis. Next, we describe more design details.

A. Object Identifier Management

The management of object identifiers includes: (1) creating an identifier for a memory object when it is allocated, (2) collecting the metadata of the object, e.g., the bounds and liveness information, and (3) attaching the identifier to the first pointer pointing to the object, e.g., a register. BSan creates identifiers using a global 64-bit counter, which is atomically increased by one each time after a new identifier is generated. That is, BSan *never* reuses previous identifiers. To maintain the collected metadata of memory objects, BSan creates a hash table called *object table* using the identifier as the key. Each entry of the table corresponds to a memory object, and when a memory object is deallocated, the corresponding entry will be updated to indicate the object is not valid for access.

1) Heap Objects: Typically, heap objects are allocated and deallocated explicitly by invoking standard library routines, e.g., `malloc()` and `free()`, or system calls, e.g., `mmap()` and `munmap()` on Linux. By intercepting invocations to these routines and system calls, BSan can collect the metadata of allocated heap objects, e.g., starting addresses and sizes. Once a heap object is successfully allocated, BSan creates a new identifier for the object and attaches the identifier to the register that returns the pointer of the object, e.g., `rax` of x86-64 and `x0` of AArch64. Also, BSan offers a flexible programming interface for users to specify customer memory management routines. This allows BSan to capture application-specific heap object allocations and deallocations.

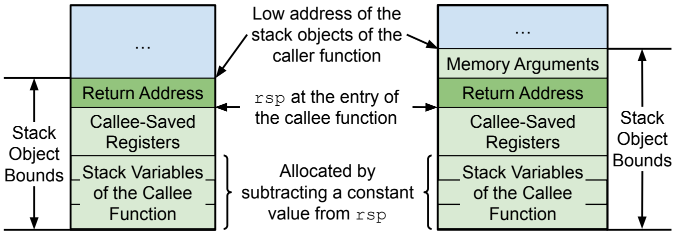

...
Return Address
Callee-Saved Registers
Stack Variables of the Callee Function
Stack Object Bounds
Low address of the stack objects of the caller function
rsp at the entry of the callee function
Allocated by subtracting a constant value from rsp
...
Memory Arguments
Return Address
Callee-Saved Registers
Stack Variables of the Callee Function
Stack Object Bounds

Fig. 2. BSan collects the stack object bounds information based on the stack structure. The left and right sides show x86-64 stacks without and with memory arguments, respectively.

2) Stack Objects: Different from heap objects, stack objects are allocated and deallocated *implicitly*, which renders it challenging to monitor their allocations and deallocations. To address this issue, BSan considers function calls and returns[2] as indicators of allocating and deallocating stack objects, respectively, inspired by the observation that stack objects are typically allocated in stack frames of functions. Specifically, BSan creates an identifier for stack objects at the entry of a function and attaches the identifier to the stack pointer register, e.g., `rsp` of x86-64 and `sp` of AArch64. The old identifier attached to the stack pointer register, which was created for the caller function, is saved and recovered when the current function returns. Note that, due to the lack of the type information in binaries, BSan only assigns a *single* identifier for all stack objects residing in the same stack frame. That means BSan cannot detect intra-stack-frame buffer overflows. This is a common limitation of state-of-the-art binary-level detectors, including RetroWrite and Dr. Memory.

After an identifier is created for the stack objects in a function F, BSan needs to collect the bounds information for the identifier, i.e., determining the high and low addresses.

High Address. Intuitively, we can use the stack pointer register value V_{sp} at the entry of F as the high address. However, this may cause false alarms for memory error detection, as the accesses to *memory arguments*—passed through the stack and located in addresses higher than V_{sp}, as shown in Figure 2—will be considered as out-of-bounds accesses. To solve this issue, BSan uses the low address determined for the identifier created at the caller function of F as the high address. This way, BSan can include memory arguments in the bounds. Since this will also include the return address stored on the stack, BSan marks down the location of the return address and reports a memory error if it is overwritten by a memory access instruction.

[2] In general, function calls and returns can be identified through corresponding instructions, e.g., `call` and `ret` of x86-64. However, the tail-call optimization may complicate this process. BSan addresses this issue by detecting tail-call-related branch instructions using state-of-the-art techniques [14], [22], [34] and treating such branches in a similar way to function calls and returns.

$$\frac{Init}{\forall x\, ID[x] \leftarrow Null}$$

$$\frac{Reg \leftarrow Constant}{ID[Reg] \leftarrow Null} \quad \frac{Mem \leftarrow Constant}{ID[Mem] \leftarrow Null} \quad \frac{Reg \leftarrow MemoryAllocation}{ID[Reg] \leftarrow New\_Id}$$

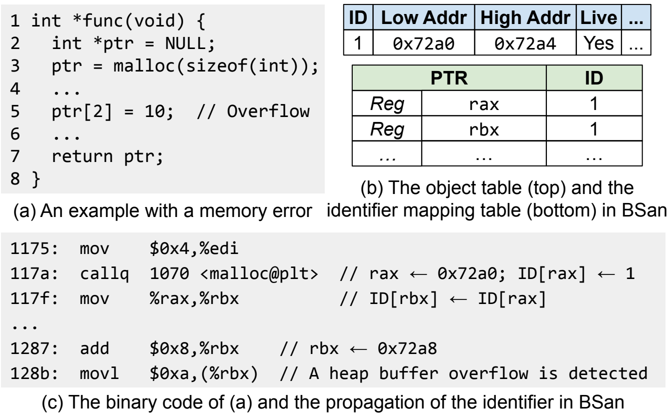
$$\frac{Reg \leftarrow Mem}{ID[Reg] \leftarrow ID[Mem]} \quad \frac{Mem \leftarrow Reg}{ID[Mem] \leftarrow ID[Reg]} \quad \frac{Reg \leftarrow \text{UnaryOp } X}{ID[Reg] \leftarrow ID[X]}$$

$$\frac{Reg \leftarrow X \text{ BinaryOp } Y \quad ID[X] = Null \quad ID[Y] = Null}{ID[Reg] \leftarrow Null}$$

$$\frac{Reg \leftarrow X \text{ BinaryOp } Y \quad ID[X] \neq Null \quad ID[Y] = Null}{ID[Reg] \leftarrow ID[X]}$$

$$\frac{Reg \leftarrow X \text{ BinaryOp } Y \quad ID[X] \neq Null \quad ID[Y] \neq Null}{ID[Reg] \leftarrow Null}$$

Fig. 3. Propagation rules of object identifiers in BSan.



```
1 int *func(void) {
2   int *ptr = NULL;
3   ptr = malloc(sizeof(int));
4   ...
5   ptr[2] = 10;  // Overflow
6   ...
7   return ptr;
8 }
```

(a) An example with a memory error

ID	Low Addr	High Addr	Live	...
1	0x72a0	0x72a4	Yes	...

PTR		ID
Reg	rax	1
Reg	rbx	1
...	...	...

(b) The object table (top) and the identifier mapping table (bottom) in BSan

```
1175:  mov    $0x4,%edi
117a:  callq  1070 <malloc@plt>  // rax ← 0x72a0; ID[rax] ← 1
117f:  mov    %rax,%rbx          // ID[rbx] ← ID[rax]
...
1287:  add    $0x8,%rbx    // rbx ← 0x72a8
128b:  movl   $0xa,(%rbx)  // A heap buffer overflow is detected
```

(c) The binary code of (a) and the propagation of the identifier in BSan

Fig. 4. An example showing how BSan detects a heap buffer overflow.

Low Address. BSan determines the low address based on the observation that the stack space is usually allocated by subtracting a constant value from the stack pointer register in the prologue of a function. The constant value indicates the actual size of the allocated stack space. By identifying and instrumenting such a subtraction instruction in F, BSan can calculate the low address. Note that it is possible that there is no such a subtraction instruction in a function because, for example, the function has no stack variable, or all stack variables are promoted to registers by the compiler. In that case, BSan simply counts the number of callee-saved registers saved to the stack, e.g., through the `push` instruction of x86-64, in the current function to calculate the low address. In addition, BSan also monitors the invocations to the `alloca()` routine, which dynamically allocates memory on the stack, to adjust the low address if necessary.

3) Global Objects: BSan creates a single identifier for global objects located in the data section of an executable binary, which is loaded into the memory at the beginning of the execution. BSan uses the start address and the size of the loaded data section to calculate the high and low addresses for the identifier. BSan attaches the identifier to the program counter (PC) register, e.g., `rip` of x86-64, as most pointers pointing to global objects are derived from the PC register.

B. Global Identifier Propagation

BSan creates a disjoint data structure, called *identifier mapping table*, to maintain the identifiers attached to registers and memory locations that contain pointers. After an identifier is created for an object, it is initially attached to the first pointer that points to the object, e.g., `rax` of x86-64 used to return the address of an allocated heap object. Then, BSan propagates the identifier in the following execution by dynamically instrumenting executed instructions.

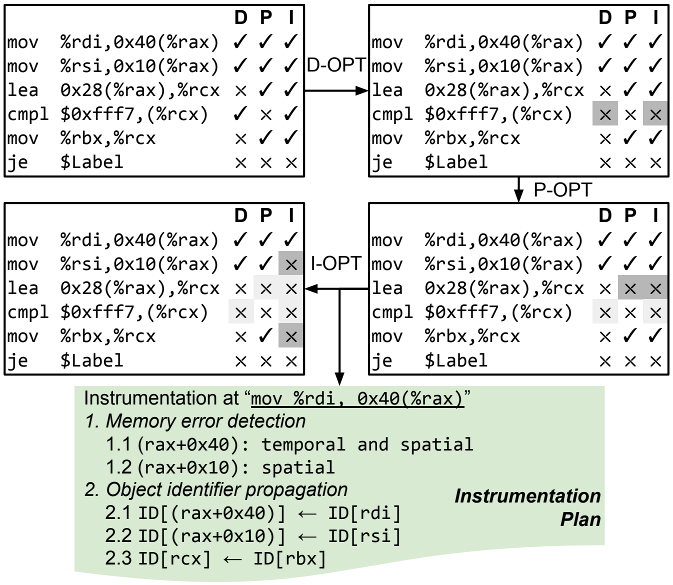
BSan propagates identifiers according to instruction types. In general, there are two types of instructions that may produce a new pointer from an existing pointer. First, a data movement instruction that copies a pointer between two registers or a register and a memory location. Second, an arithmetic/logic instruction that calculates a new pointer from an old pointer through an arithmetic/logic operation. For both types, BSan propagates the identifier attached to the old pointer to the new pointer by updating the identifier mapping table. Note that BSan does not perform any memory error detection on the new pointer until it is dereferenced to access the memory. In other words, it is allowed to produce an invalid pointer if the pointer is never used for any memory access. Figure 3 shows the propagation rules used by BSan.

In addition to the two types of instructions mentioned above, BSan also pays special attention to other instructions that may involve pointers. In particular, if an instruction computes a result from two pointers and both of them have identifiers, BSan will *not* propagate any of the identifiers to the computation result. This is inspired by the observation that the computation result is typically not a new pointer but the offset between the two pointers. Thus, it should not have an identifier. It is also worth pointing out that BSan does *not* need to take extra care of function call/return instructions. The reason is that for both caller-saved and callee-saved registers if their values need to be preserved across a function call, the compiler would generate necessary instructions to spill and restore them. Hence, BSan only needs to propagate identifiers for those instructions accordingly.

C. Memory Error Detection

BSan performs memory error detection before a memory access instruction is executed. The detection process includes three steps. (1) BSan uses the base register (or the index register) in the memory operand of the instruction to look up the identifier mapping table to get the identifier attached to the register. (2) BSan uses the identifier to search the object table to obtain the metadata information of the corresponding object. (3) BSan checks the accessed memory address against the metadata to detect spatial and temporal memory errors. If no error is detected, the instruction proceeds to access the memory. Otherwise, BSan will report a memory error.

Figure 4 uses an example to show how a memory error is detected by BSan. As shown in the source code, there is an out-of-bounds access to the heap object allocated at line 3. BSan instruments the binary code and assigns the identifier 1 to the object after it is successfully allocated, i.e., after the `callq` instruction. It also updates the identifier mapping



	D	P	I
`mov %rdi,0x40(%rax)`	✓	✓	✓
`mov %rsi,0x10(%rax)`	✓	✓	✓
`lea 0x28(%rax),%rcx`	×	✓	✓
`cmpl $0xfff7,(%rcx)`	✓	×	✓
`mov %rbx,%rcx`	×	✓	✓
`je $Label`	×	×	×

	D	P	I
`mov %rdi,0x40(%rax)`	✓	✓	✓
`mov %rsi,0x10(%rax)`	✓	✓	✓
`lea 0x28(%rax),%rcx`	×	✓	✓
`cmpl $0xfff7,(%rcx)`	×	×	×
`mov %rbx,%rcx`	×	✓	✓
`je $Label`	×	×	×

	D	P	I
`mov %rdi,0x40(%rax)`	✓	✓	✓
`mov %rsi,0x10(%rax)`	✓	✓	✓
`lea 0x28(%rax),%rcx`	×	×	×
`cmpl $0xfff7,(%rcx)`	×	×	×
`mov %rbx,%rcx`	×	✓	✓
`je $Label`	×	×	×

	D	P	I
`mov %rdi,0x40(%rax)`	✓	✓	✓
`mov %rsi,0x10(%rax)`	✓	✓	×
`lea 0x28(%rax),%rcx`	×	×	×
`cmpl $0xfff7,(%rcx)`	×	×	×
`mov %rbx,%rcx`	×	✓	×
`je $Label`	×	×	×

```
Instrumentation at "mov %rdi, 0x40(%rax)"
1. Memory error detection
   1.1 (rax+0x40): temporal and spatial
   1.2 (rax+0x10): spatial
2. Object identifier propagation
   2.1 ID[(rax+0x40)] ← ID[rdi]
   2.2 ID[(rax+0x10)] ← ID[rsi]
   2.3 ID[rcx] ← ID[rbx]
```

Fig. 5. BSan conducts three static analysis passes to generate an instrumentation plan to optimize the performance of online memory error detection. "**D**": memory error detection; "**P**": object identifier propagation; "**I**": instruction instrumentation; "✓": the corresponding operation is required; "×": the corresponding operation is optimized out by the static analysis.

table to indicate that the identifier attached to the register `rax` is 1, as the pointer of the allocated object is returned through `rax`. Next, when the pointer in `rax` is copied to `rbx`, BSan propagates the identifier to `rbx` by updating the identifier mapping table. The `add` instruction does not change the identifier attached to `rbx`. Finally, when the pointer in `rbx` is dereferenced, BSan reports a memory error, as the address in `rbx` is not within the bounds of the object indicated by the identifier attached to `rbx`.

D. *Offline Analysis*

To optimize the performance efficiency of the online detection part, particularly the high overhead incurred by propagating object identifiers, BSan conducts an offline static analysis on the target binary. The analysis essentially runs the following three passes on the control-flow graph (CFG) of each function in the target binary. (1) The *detection optimization* (D-OPT) pass checks the necessity of performing memory error detection on memory access instructions and removes redundant detection. For example, if a memory object has been checked at a previous instruction, the checks at following instructions can be skipped if they are guaranteed to be safe. (2) The *propagation optimization* (P-OPT) pass examines the necessity of propagating identifiers for instructions based on the analysis result of the D-OPT pass and removes unnecessary propagation, e.g., a propagated identifier is never checked for memory error detection. (3) Based on the analysis results of the previous two passes, the *instrumentation optimization* (I-OPT) pass aims to minimize dynamically instrumented instructions and generates an instrumentation plan to instruct the following dynamic instrumentation.

To understand this process, let us consider the example in Figure 5. In this example, without the offline static analysis, BSan needs to instrument the first *five* instructions to perform memory error detection and object identifier propagation. To reduce the performance overhead, BSan first invokes the D-OPT pass. Through this pass, BSan can identify that the memory error detection for the `cmpl` instruction is redundant. The reason is that it accesses the same object as the previous two memory access instructions, and the accessed bounds are within the bounds accessed by the previous instructions. That means, if the previous instructions do not cause any memory error, it will not also. Note that BSan *separately* determines the necessity of temporal and spatial error detection for a memory access instruction. Next, the P-OPT pass is conducted. With this pass, BSan finds out that the identifier propagation for the `lea` instruction can be removed, given that the destination register `rcx` is not used for memory error checking. Finally, the I-OPT pass combines the remaining memory error detection and identifier propagation to produce an instrumentation plan, which indicates that only the first instruction needs to be instrumented. Note that BSan also includes the address of each source instruction that requires memory error detection in the instrumentation plan for memory error reporting.

It is worth pointing out that the offline static analysis in BSan is *conservative* in the sense that it only removes the detection/propagation for an instruction if the static analysis can completely ensure the safety of the removal. That means BSan may miss optimization opportunities due to the natural inaccuracy of the static analysis, e.g., the aliasing between different pointers. But, it will *not* miss any memory errors because of applying the static analysis.

V. Evaluation

We have implemented a research prototype of BSan using Dyninst [5] (version 10.1.0, for offline analysis) and DynamoRIO [7] (version 9.0.1, for online detection). The prototype currently supports x86-64 binaries. The source code of the prototype is publicly available on GitHub.

Experimental Methodology. Our evaluation aims to answer two key questions. (1) **Detection Effectiveness**. Can BSan detect memory errors in binaries and outperform state-of-the-art binary-level detectors in terms of reporting more memory errors? (2) **Detection Efficiency**. How much performance and memory overheads does BSan introduce, and are the overheads comparable to existing binary-level detectors?

To this end, we conduct comprehensive experiments to evaluate BSan, covering a wide range of benchmark suites and real-world applications. More specifically, we evaluate the detection effectiveness of BSan using the Juliet Test Suite [19] and a list of 22 real-world memory errors, which were found from real-world applications and the common vulnerabilities and exposures (CVE) website [13]. Some of these memory errors were also used by previous research work for evaluation. Juliet contains a collection of test cases under different common weakness enumerations (CWEs). Since BSan only detects

TABLE I

Detection results of Juliet test cases. For each specific memory error, Juliet includes two test cases with and without the error, respectively. The "**#Cases**" column shows the total number of test cases with and without memory errors in each CWE category. "**FN**": false negative rate; "**FP**": false positive rate. *The detection results of MTSan are from the MTSan paper [10].

CWE-ID	**Description**	**#Cases**	**Dr. Memory**		**Memcheck**		**RetroWrite**		**MTSan***		**BSan**	
			FN	**FP**	**FN**	**FP**	**FN**	**FP**	**FN**	**FP**	**FN**	**FP**
121	Stack Buffer Overflow	3100+3100	80.58%	0	71.39%	0	50.39%	0	44.65%	0	**44.48%**	0
122	Heap Buffer Overflow	3870+3870	34.01%	0	34.37%	0	18.53%	0	19.69%	0	**14.75%**	0
124	Buffer Underflow	1168+1168	45.55%	0	32.28%	0	32.21%	0	**1.46%**	0	18.32%	0
126	Buffer Over-Read	870+870	53.56%	0	55.17%	0	52.87%	0	53.10%	0	**28.32%**	0
127	Buffer Under-Read	1168+1168	52.48%	0	53.51%	0	37.33%	0	9.85%	0	**8.31%**	0
415	Double Free	818+818	0	0	0	0	0	0	0	0	0	0
416	Use After Free	393+393	0	0	0	0	0	0	3.05%	0	0	0
Total		11387+11387	47.64%	0	37.34%	0	30.59%	0	24.17%	0	**20.71%**	0

memory errors, we focus our evaluation on memory-error-related CWEs. To evaluate the detection efficiency of BSan, we use the SPEC CPU 2017 benchmark suite [12] for single-threaded performance, the PARSEC benchmark suite [6] for multi-threaded performance, and a set of 10 real-world applications from different domains, such as PDF reader, XML language parsing, image library, audio processing, etc.

We also compare BSan with four state-of-the-art binary-level detectors, including DynamoRIO-based Dr. Memory [8], Valgrind [27]-based Memcheck [32], RetroWrite [15], and MTSan [10]. Since the source code of MTSan is unavailable[3], we cannot evaluate it on our platform. Fortunately, MTSan was also evaluated with the Juliet Test Suite and some CVE vulnerabilities used in our experiments [10]. Thus, we reuse the data presented in the MTSan paper for the comparison. Note that we cannot evaluate all bugs used in the MTSan paper because of the lack of bug-triggering inputs. For RetroWrite, due to the brittleness of static binary rewriting, we encountered various errors when running it in our experiments, though we have carefully selected the same compiler version and options mentioned in the paper [15]. We also contacted the authors of RetroWrite, but the communication did not lead to any solutions. As a result, we can only present results for experiments RetroWrite can successfully run.

The experimental platform is equipped with an octa-core Intel i9-9900 CPU running at 3.10GHz and 64GB of main memory. The operating system is Ubuntu 20.04 with the Linux kernel (version 5.4.0). We use GCC (version 9.4.0) to compile test programs. The platform is occupied exclusively during our experiments. In addition, we run each program five times and use the average execution time as its final performance.

A. Detection Effectiveness

We first present experimental results to demonstrate the detection effectiveness of BSan.

1) Juliet: Table I shows the detection results of Juliet test cases. As we can see, none of the evaluated detectors reports false positives. This is because these detectors monitor program execution to detect memory errors, and thus have a very low chance to report false positives. Regarding false negatives, BSan outperforms other detectors in four CWE categories, i.e., CWE-121, CWE-122, CWE-126, and CWE-127. That is, BSan can detect more memory errors than other detectors in these CWE categories. For memory errors in the two CWE categories, i.e., CWE-415 and CWE-416, BSan does not miss any of them. This aligns with Dr. Memory, Memcheck, and RetroWrite, but is better than MTSan, which misses some memory errors. For CWE-124, MTSan achieves the best detection result, though BSan can detect more memory errors than the other three detectors. Our further study shows that MTSan tries to guess object boundaries *during the fuzzing process*, allowing it to occasionally detect some intra-buffer memory errors. Since we run each test case only once, we believe that it is extremely difficult, if not impossible, to recover accurate object boundaries with a single run.

We also find that the major reason why the evaluated detectors, including BSan, cannot detect some missed memory errors is the lack of the data type information in binary code, which makes it hard to detect intra-heap-buffer and intra-stack-frame memory errors. Apart from this reason, Dr. Memory and Memcheck have very limited support for detecting memory errors related to stack objects, e.g., CWE-121. Due to the shadow-memory-based detection approach, Dr. Memory, Memcheck, and RetroWrite can miss many memory errors, e.g., CWE-122, CWE-126, and CWE-127. In addition, RetroWrite suffers from various static binary rewriting errors, which can either fail the static rewriting process or produce binaries that behave differently compared to the original binaries, e.g., exhibiting infinite executions and segmentation faults.

In summary, the detection results of Juliet test cases show that BSan can effectively detect memory errors. For four CWEs, it outperforms state-of-the-art binary-level detectors.

2) Real-World Memory Errors: Table II shows the detection results of the 22 real-world memory errors. As the table shows, BSan successfully detects all memory errors. In contrast, state-of-the-art detectors cannot detect many of them due to different reasons. Dr. Memory and Memcheck miss stack-based memory errors due to their limited support for stack

[3] The link in the MTSan paper https://github.com/vul337/mtsan-repo is not accessible. We also contacted the authors but did not receive any response.

TABLE II

DETECTION RESULTS OF REAL-WORLD MEMORY ERRORS AND CVE VULNERABILITIES. "**OPT**": COMPILER OPTIMIZAITON LEVEL; "**BINSIZE**": BINARY SIZE MEASURED IN KILOBYTES (KB); "**DM**": DR. MEMORY; "**MC**": MEMCHECK;"**RW**": RETROWRITE; "**MT**": MTSAN; "**BS**": BSAN. "SW": STACK OUT-OF-BOUNDS WRITE; "HW": HEAP OUT-OF-BOUNDS WRITE; "HR": HEAP OUT-OF-BOUNDS READ; "HF": HEAP USE-AFTER-FREE; "SF": STACK USE-AFTER-FREE; "F": STATIC BINARY REWRITING FAILURE; "-": NOT EVALUATED. *DETECTION RESULTS OF MTSAN ARE FROM THE PAPER [10].

	Type	OPT	BinSize	DM	MC	RW	MT*	BS
CVE-2018-20004	SW	O0	121.17	×	×	F	✓	✓
		O1	175.80	×	×	F	✓	✓
		O2	178.82	×	×	F	✓	✓
		O3	327.66	×	×	F	✓	✓
CVE-2020-21676	SW	O0	1333.98	×	×	F	×	✓
		O1	2095.13	×	×	F	×	✓
		O2	2315.62	×	×	F	×	✓
		O3	3060.34	×	×	F	×	✓
CVE-2021-20294	SW	O0	4060.40	×	×	F	✓	✓
		O1	4637.23	×	×	F	✓	✓
		O2	5298.74	×	×	F	✓	✓
		O3	5613.40	×	×	F	✓	✓
CVE-2016-10270	HR	O0	63.22	✓	✓	F	✓	✓
		O1	105.71	✓	✓	F	✓	✓
		O2	112.74	✓	✓	F	✓	✓
		O3	169.19	✓	✓	F	✓	✓
CVE-2018-20005	HF	O0	139.63	✓	✓	F	✓	✓
		O1	252.61	✓	✓	F	✓	✓
		O2	283.19	✓	✓	F	✓	✓
		O3	362.68	✓	✓	F	✓	✓
CVE-2017-14408	SW	O0	104.98	×	×	×	✓	✓
		O1	118.73	×	×	×	✓	✓
		O2	118.53	×	×	×	✓	✓
		O3	150.03	×	×	×	✓	✓
CVE-2017-9047	SW	O0	131.59	×	×	×	✓	✓
		O1	203.77	×	×	×	✓	✓
		O2	216.43	×	×	×	✓	✓
		O3	225.49	×	×	×	✓	✓
CVE-2016-10271	HF	O0	63.22	✓	✓	✓	✓	✓
		O1	105.71	✓	✓	✓	✓	✓
		O2	112.74	✓	✓	✓	✓	✓
		O3	169.19	✓	✓	✓	✓	✓
CVE-2013-4243	HF	O0	34.72	✓	✓	✓	✓	✓
		O1	50.34	✓	✓	✓	✓	✓
		O2	51.97	✓	✓	✓	✓	✓
		O3	52.66	✓	✓	✓	✓	✓
CVE-2020-21675	SW	O0	1417.95	×	×	×	✓	✓
		O1	2401.31	×	×	×	✓	✓
		O2	2655.22	×	×	×	✓	✓
		O3	3066.47	×	×	×	✓	✓
CVE-2019-8356	SW	O0	126.96	×	×	F	-	✓
		O1	179.09	×	×	F	-	✓
		O2	186.41	×	×	F	-	✓
		O3	206.00	×	×	F	-	✓
CVE-2020-11528	SW	O0	16.30	×	×	✓	-	✓
		O1	21.03	×	×	✓	-	✓
		O2	22.12	×	×	✓	-	✓
		O3	22.12	×	×	✓	-	✓
CVE-2023-31981	SW	O0	519.24	×	×	F	-	✓
		O1	729.62	×	×	F	-	✓
		O2	762.37	×	×	F	-	✓
		O3	824.14	×	×	F	-	✓
Fdkaac Bug#55	SW	O0	238.41	×	×	F	-	✓
		O1	341.70	×	×	F	-	✓
		O2	432.41	×	×	F	-	✓
		O3	520.92	×	×	F	-	✓
GPAC Bug#1348	SW	O0	275.73	×	×	F	-	✓
		O1	373.38	×	×	F	-	✓
		O2	377.43	×	×	F	-	✓
		O3	385.21	×	×	F	-	✓
CVE-2022-27135	HW	O0	1234.94	✓	✓	F	-	✓
		O1	2193.06	✓	✓	F	-	✓
		O2	2498.78	✓	✓	F	-	✓
		O3	2658.07	✓	✓	F	-	✓
CVE-2023-36274	HW	O0	181.14	✓	✓	F	-	✓
		O1	191.15	✓	✓	F	-	✓
		O2	191.18	✓	✓	F	-	✓
		O3	191.18	✓	✓	F	-	✓
CVE-2020-18430	HR	O0	739.22	×	✓	F	-	✓
		O1	769.45	×	✓	F	-	✓
		O2	799.15	×	✓	F	-	✓
		O3	971.75	×	✓	F	-	✓
CVE-2023-0645	HR	O0	164.57	✓	✓	F	-	✓
		O1	195.48	✓	✓	F	-	✓
		O2	200.66	✓	✓	F	-	✓
		O3	220.88	✓	✓	F	-	✓
Vim Bug#11923	HR	O0	8530.87	✓	✓	F	-	✓
		O1	11935.98	✓	✓	F	-	✓
		O2	13287.80	✓	✓	F	-	✓
		O3	17035.57	✓	✓	F	-	✓
CVE-2023-5535	HF	O0	15122.07	✓	✓	F	-	✓
		O1	19726.23	✓	✓	F	-	✓
		O2	21462.36	✓	✓	F	-	✓
		O3	26232.16	✓	✓	F	-	✓
PH7 Bug#37	SF	O0	952.19	×	×	F	-	✓
		O1	1633.72	×	×	F	-	✓
		O2	2157.17	×	×	F	-	✓
		O3	2750.61	×	×	F	-	✓

objects, though they can find most heap-based memory errors. For RetroWrite, it can only detect one CVE vulnerability because it fails to rewrite other binaries to generate runnable binaries. Regarding MTSan, we only have limited detection results, as it is not available for us to evaluate. According to the MTSan paper, it fails to detect CVE-2020-21676, a stack out-of-bounds write, because this error modifies the stack canary, which causes the crashed execution of the application before MTSan can detect it. To summarize, we can conclude from the detection results that BSan is more effective than state-of-the-art detectors in detecting real-world memory errors.

From Table II, we can also observe that BSan is *insensitive* to different compiler optimization levels, i.e., from `O0` to `O3`. This demonstrates that BSan does *not* need to take special care of compiler optimization levels, as its design is general and not specific to any optimization levels.

```
 1  static ph7_hashmap_node *
 2  HashmapNodeMerge(ph7_hashmap_node *pA, ...) {
 3    ph7_hashmap_node result, *pTail;
 4    result.pNext = result.pPrev = 0;
 5    pTail = &result;
 6    ...
 7    if (xCmp(pA, pB, pCmpData) < 0) {
 8      pTail->pPrev = pA;
 9      pA->pNext = pTail; // The local pointer pTail
10      pTail = pA;        // escapes from the current
11      pA = pA->pPrev;    // function via pA->pNext.
12    }
13    ...
14  }
```

Fig. 6. PH7 Bug#37, a stack use-after-free bug. Note that state-of-the-art binary-level detectors cannot detect this bug due to their shadow memory-based approach, while BSan can successfully report this bug.

Figure 6 shows the simplified source code of PH7 Bug#37, a real-world stack use-after-free bug. In this bug, the local pointer `pTail` is initialized with the address of the local variable `result` at line 5. However, this pointer is later propagated out of the function through the pointer `pA->pNext` at

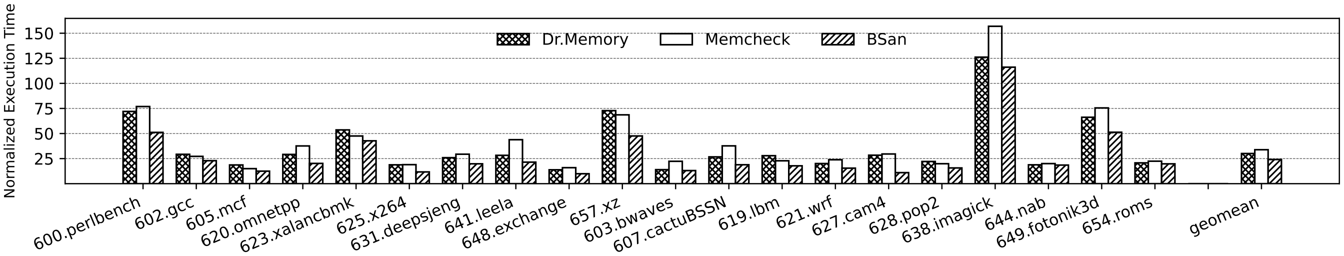


Fig. 7. Normalized execution times of SPEC CPU 2017 benchmarks with different detectors. The baseline is the native execution time without any detector. RetroWrite is excluded due to the errors it has when statically rewriting the binaries.

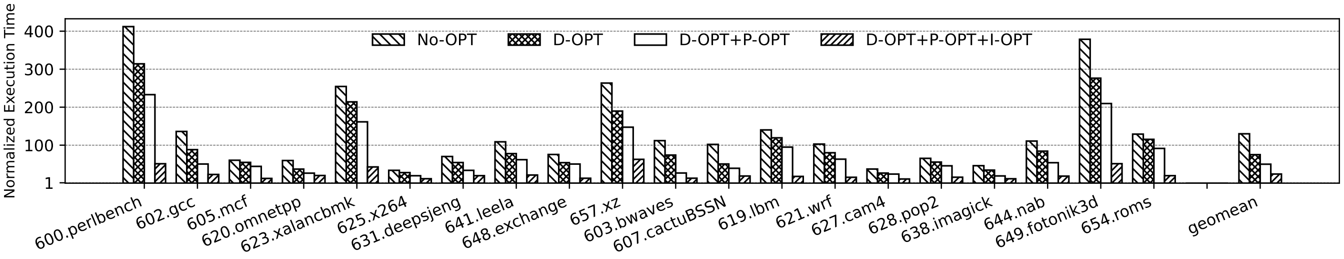


Fig. 8. Normalized execution times of SPEC CPU 2017 benchmarks with BSan using different optimizations. The baseline is the native execution time without any detector. "No-OPT": no offline optimization; "D-OPT": applying offline detection optimization only; "D-OPT+P-OPT": applying offline detection and propagation optimizations only; "D-OPT+P-OPT+I-OPT": applying all offline optimizations, i.e., detection, propagation, and instrumentation optimizations.

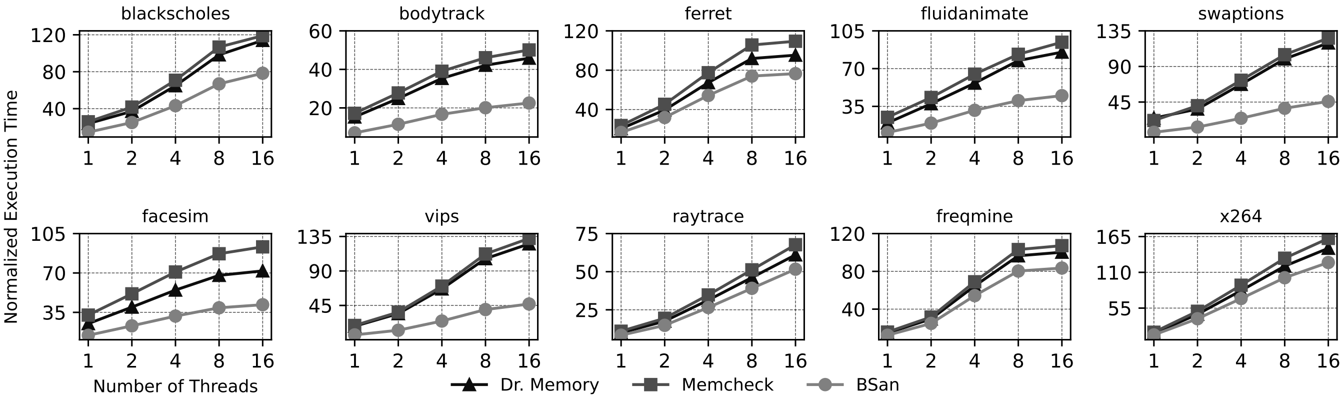


Fig. 9. Normalized execution times of PARSEC benchmarks with different detectors under different numbers of threads. The baseline is the native execution time without any detector under the corresponding thread number. RetroWrite is excluded in this experiment due to its static binary rewriting errors.

line 9. As a consequence, a stack use-after-free error will happen when `pA->pNext` is dereferenced later. Unfortunately, none of the evaluated binary-level detectors can find this bug because when `pA->pNext` is dereferenced, it points to a valid address on the stack, rendering the shadow memory-based approach ineffective. Thanks to the identifier-based approach, BSan can successfully report this bug by checking the identifier attached to `pA->pNext`, which corresponds to a deallocated stack object when this pointer is dereferenced.

B. Performance Efficiency

We next study the runtime performance of BSan and compare it with other detectors.

1) Single-Threaded Performance: Figure 7 shows the performance efficiency of SPEC CPU 2017 benchmarks running with different detectors. Here, although RetroWrite is not included because it fails to perform static binary rewriting for the benchmark binaries, it is worth noting that RetroWrite is expected to have *better* runtime performance than the detectors in the figure, including BSan, as it eliminates the high performance overhead incurred by dynamic instrumentation.

As shown in Figure 7, BSan consistently achieves the best performance efficiency for all benchmarks among the three evaluated detectors. This demonstrates the high performance efficiency of BSan compared to existing detectors. On average, BSan incurs 24.07× performance slowdown, while Dr. Memory and Memcheck result in 30.06× and 33.85× performance slowdown, respectively. Given that Dr. Memory and Memcheck are widely adopted in practice to detect memory errors in application binary code, this result shows the strong

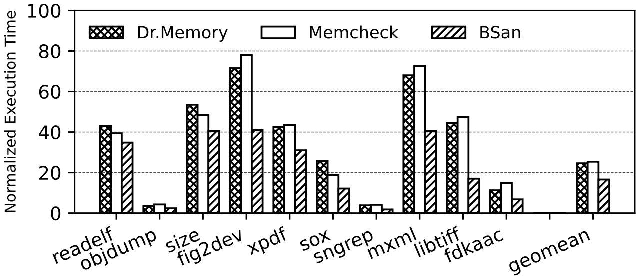


Fig. 10. Normalized execution times of real-world applications with different detectors. The baseline is the native execution time without any detector.

practicability of BSan in terms of detection efficiency.

To understand the optimization effectiveness of the offline analysis in BSan, we further measure its performance with different optimizations applied. Figure 8 shows the results. As we can see, without applying any offline optimizations, BSan introduces significant performance slowdown, 129.88× on average and as high as 412.03× for *600.perlbench*. This is because it needs to instrument most of the executed instructions to conduct memory error detection and object identifier propagation. This result echoes the necessity of developing optimizations to improve the detection efficiency. As the figure shows, with more optimizations applied, the performance slowdown is reduced substantially, from 129.88× to 75.72× with D-OPT, 49.71× with D-OPT and P-OPT, and 24.07× with all optimizations, i.e., D-OPT, P-OPT, and I-OPT. This demonstrates the effectiveness of the offline analysis in BSan in removing redundant detection and propagation.

2) Multi-Threaded Performance: Figure 9 shows the performance results of PARSEC benchmarks running with different detectors under different numbers of threads. Again, RetroWrite is not included because of its static binary rewriting errors. From the figure, we can clearly see that BSan achieves better performance efficiency than the other two detectors. Overall, the three detectors introduce 91.51× (Dr. Memory), 101.21× (Memcheck), and 55.83× (BSan) performance slowdown, respectively, when running the benchmarks with 16 threads. This shows that BSan is more applicable when detecting memory errors in multi-threaded applications.

An interesting observation we can make from Figure 9 is that as the thread number increases, the performance overhead incurred by memory error detection also increases for all three detectors. A possible explanation for this phenomenon is that memory error detection may contend with application threads for limited computing resources, e.g., CPU Cache, which can offset the performance benefit achieved by increasing the number of threads.

3) Real-World Applications: Figure 10 shows the performance efficiency of the 10 real-world applications running with different detectors. BSan again surpasses Dr. Memory and Memcheck, benefiting from its offline optimizations. On average, compared to the baseline, BSan brings in a 16.40× performance slowdown, while Dr. Memory and Memcheck lead to 24.57× and 25.37× slowdown, respectively. With this result, we are confident that it is practical to apply BSan for detecting memory errors in COTS binary code.

TABLE III
OFFLINE ANALYSIS TIMES (IN SECONDS) OF BSAN FOR SPEC CPU 2017 BENCHMARKS AND THEIR BINARY SIZES (IN KB).

	Size	**Time**		**Size**	**Time**
600.perlbench	7107	321.10	602.gcc	38035	801.48
605.mcf	97	2.09	620.omnetpp	15188	302.94
623.xalancbmk	51439	754.75	625.x264	1853	36.84
631.deepsjeng	353	5.77	641.leela	2436	33.23
648.exchange	151	1.86	657.xz	622	19.94
603.bwaves	106	1.23	607.cactuBSSN	12799	198.44
619.lbm	75	14.23	621.wrf	5574	28.49
627.cam4	7170	27.98	628.pop2	4782	24.48
638.imagick	4835	260.82	644.nab	488	15.04
649.fotonik3d	672	1.42	654.roms	2344	1.43

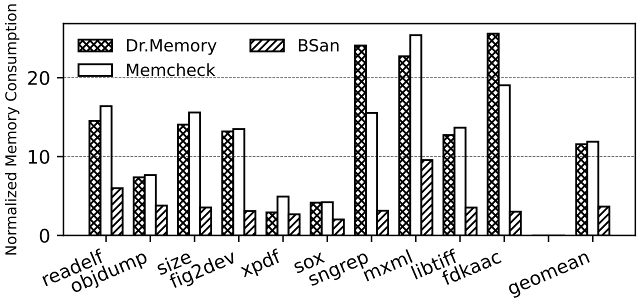


Fig. 11. Normalized memory consumption with different detectors. The baseline is the memory size used by the native execution without any detector.

C. Cost Analysis

We finally study the analysis time of the offline analysis and the memory overhead of the online detection in BSan.

1) Offline Analysis Time: Table III shows the offline analysis times for SPEC CPU 2017 benchmarks. As shown in the table, for most benchmarks, BSan can complete the offline analysis in less than 60 seconds. For some benchmarks, e.g., *602.gcc* and *623.xalancbmk*, the analysis times are longer than 10 minutes because of their large binary sizes. On average, BSan needs 89.93 seconds to analyze a binary application. We further investigate the offline analysis process and find that the three optimization passes, i.e., detection, propagation, and instrumentation, consume an average of 14.12%, 17.36%, and 15.59% of the total offline analysis time, respectively. The remaining 52.93% analysis time is used by the Dyninst infrastructure to disassemble the binaries and construct CFGs. Since the offline analysis of BSan is a one-time cost, we consider the analysis times acceptable in practice.

2) Memory Overhead of Online Detection: Figure 11 shows the normalized memory consumption of the 10 real-world applications running with different detectors. From the figure, we can clearly see that BSan incurs much less memory overhead than the other two detectors. Overall, they respectively introduce 11.56× (Dr. Memory), 11.88× (Memcheck), and 3.65× (BSan) memory overhead. The major reason for the

TABLE IV
The comparison between BSan and state-of-the-art binary-level detectors. “**DT**”: detection technique; “**IM**”: instrumentation mechanism; “**HI**”: hardware independence. “OA”: offline analysis; “SM”: shadow memory; “HF”: hardware feature; “ID”: object identifier; “DBI”: dynamic binary instrumentation; “SBR”: static binary rewriting.

	DT	**IM**	**HI**	**Error Type**		**Object Type**		
				Spatial	Temporal	Heap	Stack	Global
Undangle [9]	OA	DBI	✓	×	✓	✓	×	×
QASan [18]	SM	DBI	✓	✓	✓	✓	×	×
Dr. Memory [8]	SM	DBI	✓	✓	✓	✓	×	×
Memcheck [32]	SM	DBI	✓	✓	✓	✓	×	×
RetroWrite [15]	SM	SBR	✓	✓	✓	✓	✓	×
MTSan [10]	HF	SBR	×	✓	✓	✓	✓	✓
BSan	ID	Hybrid	✓	✓	✓	✓	✓	✓

high memory overhead of the other detectors is their adoption of the shadow-memory-based approach, which requires the allocation of a large region of shadow memory and populating it with meaningful shadow bits even if the corresponding application memory is not allocated. In contrast, BSan uses the identifier-based approach, which allows it to allocate memory for the detection metadata on demand.

VI. Related Work

Due to the high impact of memory errors and the popularity of binary code, many binary-level memory error detectors have been developed. Notable examples include Dr. Memory [8], QASan [18], Undangle [9], Memcheck [32], RetroWrite [15], and MTSan [10]. Among them, Undangle is an offline detector, which analyzes the execution traces of binary applications to detect heap use-after-free errors. QASan, Dr. Memory, and Memcheck are shadow-memory-based detectors using three different dynamic binary instrumentation frameworks, i.e., QEMU [3], DynamoRIO [7], and Valgrind [27], respectively. RetroWrite uses static binary rewriting to implement a binary version of AddressSanitizer [31], which is a source-level shadow-memory-based detector. Differently, MTSan relies on the recently introduced memory tagging extension in ARM processors to detect memory errors.

Table IV summarizes the differences between BSan and state-of-the-art binary-level memory error detectors. Particularly, BSan adopts an *identifier-based* approach for memory error detection. This allows it to detect more deeply hidden memory errors in binary code compared to existing shadow-memory-based detectors, as evidenced in our experimental results. Also, it is a software-only detector with no dependence on any hardware-specific features. Though the identifier-based approach has been implemented at the source code level [24], [25], BSan addresses several unique technical challenges of applying it to binary code, e.g., creatively combining offline analysis and online detection to reduce the high runtime overhead caused by propagating object identifiers.

In addition, the techniques developed in BSan are *not* software substitutions of prior hardware-based solutions for memory safety [30], [33], [38], [40]. For instance, HeapCheck [30] assumes the availability of source code and relies on compiler-based instrumentation to leverage the unused bits in 64-bit pointers to store pointer metadata for heap memory safety checks. As discussed in §III, this cannot work for binary code. As another example, REST [33] modifies the instruction set to add new instructions for memory error detection, and as a consequence, it has to recompile program source code to generate new binaries to utilize the added instructions.

VII. Conclusion

In this paper, we have presented BSan, a powerful software-only memory error detector for COTS binaries. The distinctive feature that differentiates it from state-of-the-art binary-level detectors is the adoption of an identifier-based approach. This enables BSan to detect more deeply hidden memory errors in binary code. The development of BSan also addresses several unique technical challenges of implementing the identifier-based approach at the binary code level. Experimental results show that BSan can achieve desirable detection effectiveness. Moreover, the detection efficiency of BSan, including both performance and memory overheads, is comparable to existing dynamic instrumentation-based detectors.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the U.S. National Science Foundation under grants CNS-2238264, CNS-2330752, and CNS-2401873. This work was also partially supported by the M. G. Michael Award funded by the Franklin College of Arts and Sciences at the University of Georgia and a faculty startup funding offered by the University of Georgia.

References

[1] ARM. Armv8.5-A Memory Tagging Extension White Paper, Accessed: August 2024. https://documentation-service.arm.com/static/624ea580caabfd7b3c13e23f.

[2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP ’21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[4] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’06, page 158–168, New York, NY, USA, 2006. Association for Computing Machinery.

[5] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE ’11, page 9–16, New York, NY, USA, 2011. ACM.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT ’08, page 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, page 265–275, USA, 2003. IEEE Computer Society.

[8] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 213–223, USA, 2011. IEEE Computer Society.

[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-Free and Double-Free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 133–143, New York, NY, USA, 2012. ACM.

[10] Xingman Chen, Yinghao Shi, Zheyu Jiang, Yuan Li, Ruoyu Wang, Haixin Duan, Haoyu Wang, and Chao Zhang. MTSan: A Feasible and Practical Memory Sanitizer for Fuzzing COTS Binaries. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 841–858, Anaheim, CA, August 2023. USENIX Association.

[11] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 271–284, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Standard Performance Evaluation Corporation. SPEC CPU 2017, Accessed: 2017. https://www.spec.org/cpu2017/.

[13] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE)), Accessed: August 2024. https://www.cve.org.

[14] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[16] The Linux Kernel documentation. Memory Layout on AArch64 Linux, Accessed: August 2024. https://www.kernel.org/doc/html/latest/arch/arm64/memory.html.

[17] The Linux Kernel documentation. x86_64 Memory Management, Accessed: August 2024. https://www.kernel.org/doc/html/latest/arch/x86/x86_64/mm.html.

[18] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30, 2020.

[19] NSA Center for Assured Software. Juliet C/C++ 1.3, Accessed: October 2017. https://samate.nist.gov/SARD/test-suites/112.

[20] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. Will You Still Compile Me Tomorrow? Static Cross-Version Compiler Validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 191–201, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[22] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 24–35, New York, NY, USA, 2016. Association for Computing Machinery.

[23] Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, Accessed: August 2024. 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.

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 245–258, New York, NY, USA, 2009. Association for Computing Machinery.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.

[26] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 128–139, New York, NY, USA, 2002. Association for Computing Machinery.

[27] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[28] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851, 2021.

[29] The Chromium Projects. Memory safety, Accessed: August 2024. https://www.chromium.org/Home/chromium-security/memory-safety.

[30] Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. HeapCheck: Low-cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.*, 19(1), Jan 2022.

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[32] Julian Seward and Nicholas Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[33] Kanad Sinha and Simha Sethumadhavan. Practical Memory Safety with REST. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 600–611. IEEE Press, 2018.

[34] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static Binary Rewriting without Supplemental Information: Overcoming the Tradeoff between Coverage and Correctness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 52–61, 2013.

[35] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery.

[36] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[37] Wenwen Wang. MLEE: Effective Detection of Memory Leaks on Early-Exit Paths in OS Kernels. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 31–45. USENIX Association, July 2021.

[38] Shengjie Xu, Wei Huang, and David Lie. In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Granularity Protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, page 117–126, New York, NY, USA, 2004. Association for Computing Machinery.

[40] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-FAT: Architectural Support for Low Overhead Memory Safety Checks. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 916–929. IEEE Press, 2021.