Shadow State Encoding for Efficient Monitoring of Block-level Properties

Kostyantyn Vorobyov Julien Signoles Nikolai Kosmatov
CEA, List, Software Reliability and Security Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract

Memory shadowing associates addresses from an application's memory to values stored in a disjoint memory space called shadow memory. At runtime shadow values store metadata about application memory locations they are mapped to. Shadow state encodings – the structure of shadow values and their interpretation – vary across different tools. Encodings used by the state-of-the-art monitoring tools have been proven useful for tracking memory at a byte-level, but cannot address properties related to memory block boundaries. Tracking block boundaries is however crucial for spatial memory safety analysis, where a spatial violation such as out-of-bounds access, may dereference an allocated location belonging to an adjacent block or a different struct member.

This paper describes two novel shadow state encodings which capture block-boundary-related properties. These encodings have been implemented in E-ACSL – a runtime verification tool for C programs. Initial experiments involving checking validity of pointer and array accesses in computationally intensive runs of programs selected from SPEC CPU benchmarks demonstrate runtime and memory overheads comparable to state-of-the-art memory debuggers.

Keywords Memory safety, Shadow memory, Runtime monitoring, Frama-C

1. Introduction

Memory errors such as buffer overflows are among the most widespread and critical errors that can lead to serious defects in software [8] and account for about 50% of reported security vulnerabilities [39]. The importance of detecting all overflows in a program cannot be underestimated. This is especially the case in safety critical systems, where a single vulnerability may cost human lives.

One of the most popular techniques for detecting buffer overflows at runtime are dynamic memory analysers (or memory debuggers). These tools instrument programs with statements that monitor the program's execution at runtime. This added functionality typically records allocated memory, checks memory accesses and detects memory errors before they occur. Memory debuggers, such as Rational Purify [16], AddressSanitizer [31], MemorySanitizer [36], Dr. Memory [6], MemCheck [33] are increasingly popular. They are used in large-scale projects [4, 38] and even deployed at an operating system level [5].

The majority of the state-of-the-art monitoring tools use *memory shadowing* to keep track of memory allocated by a program at runtime and detect memory issues. In its typical use memory shadowing associates addresses from an application's memory to values stored in a disjoint memory space called shadow memory. During a program run shadow values store metadata about the memory locations they are mapped to. A powerful feature of memory shadowing is its ability to characterise every address from an application's memory space and provide fast, constant-time access to its shadow value. This feature makes shadow memory particularly attractive for dynamic memory safety analysis [6, 16, 31, 33, 36].

Shadow state encodings – the structure of shadow values and their interpretation – vary across different tools, but they are similar in that they use shadow values to store bit-level states of individual bits or bytes in an application's memory. Lift [29], and Panorama [42], tools concerned with detection of information leakage at runtime, use one bit to tag each addressable byte from application memory as public or private. Dr. Memory [6] and Purify [16] memory debuggers shadow one byte by two bits which track that byte's allocation and initialization status. MemCheck [33] and MemorySanitizer [36] use bit-to-bit shadowing to track initialization status of every bit. AddressSanitizer [31] customizes memory allocation to ensure that memory blocks are allocated at an 8-byte boundary, and tracks aligned 8-byte sequences by 1 shadow byte.

Byte-level tracking enables fast lookups and reduces runtime overheads of memory tracking which makes memory debuggers usable in practice. However, byte-level representation of a program's memory state is imprecise because it fails to capture *block-level properties*, i.e. properties related to the bounds of memory allocation units often referred to as *memory blocks*. As a consequence, tools tracking memory only at a byte-level cannot detect buffer overflows that access allocated memory.

Consider the following code snippet, where execution of the assignment at Line 2 leads to a buffer overflow.

```
1   char buf1 [1], buf2 [1];
2   buf1[1] = '0'; // potentially modifies buf2[0]
```

In a typical execution, `buf1` and `buf2` are allocated one after another. The assignment at Line 2 therefore may write to allocated memory belonging to a different buffer – `buf2`. A dynamic analyser tracking such execution observes a program modifying allocated memory and does not raise an alarm.

In all fairness, the state-of-the-art memory debuggers based on shadow memory are not completely helpless in this situation. To protect against such violations they use *red zones* [2, 15, 31] a technique that puts gaps between allocated blocks. Red zones are effective for detecting off-by-one errors, but they cannot identify all overflows. For instance, AddressSanitizer that uses red zones for

[Copyright notice will appear here once 'preprint' option is removed.]

stack monitoring detects a buffer overflow at Line 2 in the above code snippet, but may fail to detect an error for `buf1[3] = '0'`. This is because `buf1[1]` hits a red zone surrounding `buf1`, whereas `buf1[3]` accesses a memory location belonging to `buf2`.

An analogous but more dangerous situation may occur when a buffer overflow can be exploited. A carefully crafted input can be used to modify allocated memory and take control of the program without ever being detected by a memory debugger. An example of this type of violation (assuming that `i` is controlled via program inputs) is shown via a code snippet below.

```
1    void foo (int *buf, int i, int c)
2      { *(buf + i) = c; }
```

Detecting illegal accesses to allocated memory can alternatively be enabled by techniques that use lookup tables to capture per-block metadata in their entries [3, 10, 17, 20, 22, 30, 40, 41]. However, since a table lookup is typically more costly than a shadow memory access, these techniques are known to incur excessive runtime overheads when applied to memory safety problems [17, 22, 40].

Fast, block-level memory analysis can potentially be enabled using METAlloc [14], a shadow memory technique utilizing a *page table* – a shadow array of references to per-block metadata. METAlloc builds upon modern heap [13] and stack [23] organizations, relying on a fixed non-trivial common alignment of memory blocks within a page. Enforcing such a requirement may lead to changes of a program's memory layout and increase stack usage.

This paper addresses the question of sound yet practical detection of out-of-bounds violations and presents two novel shadow state encoding schemes capable of tracking memory allocated by a program at runtime with block-level precision. These schemes, called *segment-based* and *offset-based* encodings, are suitable for use with either source-code or binary-level analysis and require no runtime environment customization, compiler modifications, alignment of stack or global variables or changes to standard memory allocation facilities.

Segment-based encoding represents allocated memory blocks by fixed-length *segments* (i.e., contiguous memory regions) and relies on a common alignment bound shared by all segments. Each application segment is tracked by a corresponding segment in the shadow memory whose address is found using modulo operation.

Offset-based encoding allows tracking unaligned allocations and does not require changes to a program's layout or compiler modifications. This scheme uses a combination of two shadow spaces called *Primary* and *Secondary* shadows, such that a byte in the *Primary* shadow stores an offset relative to a *Secondary* shadow location capturing address metadata.

The proposed encodings have been implemented in E-ACSL [9] – a runtime verification tool for C programs. Additionally to tracking allocation and initialization status of individual bytes (as is common with many memory monitors), E-ACSL can detect block-level out-of-bounds accesses. Experimentation with programs selected from SPEC CPU benchmarks [35] shows that for the purpose of verifying validity of pointer and array accesses the runtime overheads of E-ACSL are comparable to the runtime overheads of such state-of-the-art memory debuggers as MemCheck [33] or Dr. Memory [6], whereas memory overheads of the proposed encodings are lower. Otherwise, for an analogous performance cost E-ACSL equipped with the proposed shadow state encodings can discover a larger class of memory safety issues consuming less memory.

The contributions made by this paper are therefore as follows:

- *Two novel shadow state encoding schemes* allowing capture of byte- and block-level properties. The key feature of the proposed approach is a fast constant-time computation allowing to identify bounds and a byte-length of the memory block a given address belongs to.
- *Proof-of-concept implementation* of the proposed approach using E-ACSL plugin of the Frama-C [21] framework for source code analysis.
- *Empirical evaluation* of the proposed technique using computationally intensive programs selected from SPEC CPU [35] datasets for CPU testing. This evaluation compares performance overheads of E-ACSL to the overheads incurred by MemCheck [33], AddressSanitizer [31], Dr. Memory [6] memory debuggers and a previous implementation of E-ACSL using patricia trie. The results of the experiments show that the proposed technique leads to runtime overheads comparable to those of state-of-the-art memory debuggers, uses less memory, but has an additional benefit of tracking memory at a block-level. Furthermore, E-ACSL maintains the ability to track a large class of memory violations with respect to the previous implementation of E-ACSL, while significantly improving its performances.

The rest of this paper is organized as follows. Section 2 gives background details relevant to memory allocation. Sections 3 and 4 describe the proposed shadow state encodings. Section 5 discusses implementation of the proposed techniques using E-ACSL plugin of Frama-C and Section 6 presents experimental results. Finally, Section 7 reviews related work and Section 8 gives concluding remarks and outlines directions of future work.

2. Memory Allocation

Shadow techniques presented in this paper apply to runtime environments which manage memory through static, dynamic and automatic allocation of disjoint memory blocks in the virtual memory space of a computer process, where random access memory is represented by a contiguous array of memory cells with byte-level addressing. Such a virtual memory space is partitioned into contiguous memory segments, comprising text, stack, heap, bss, data and potentially other segments.

The *byte-width* of a memory address, denoted A_W, refers to a number of bytes sufficient to uniquely represent a memory address in the virtual memory space of a process. Numeric memory addresses are written using hexadecimal numbers prefixed by 0x to distinguish them from decimal numbers.

A *memory block* B is represented by a contiguous memory region described by its *bounds*, that is, its start (or *base*) address $base_B$ and its end address end_B. The *length* of block B, denoted $Length_B$, is equal to $end_B - base_B + 1$ bytes. It is assumed that any memory block contains at least one byte, i.e. $base_B \leq end_B$. If address a belongs to block B (that is, $base_B \leq a \leq end_B$), then $baseoff_a = a - base_B$ is called the *byte offset* of a (with respect to the base address of B, also denoted $base_a$ in this case).

Any property involving the bounds of a memory block B, its length or the byte offset of an address belonging to B is referred to as a *block-level* property.

A memory block B is said to be *aligned* at a boundary of n, if $base_B$ is divisible by n. Block B is said to be *padded* with n bytes, if $base_B$ is preceded by at least n bytes of unusable memory.

Static or global memory allocation refers to memory blocks allocated at compile-time. Automatic memory allocation refers to memory blocks allocated and deallocated on a program's stack at runtime. In a typical program automatic blocks are unaligned or aligned at a boundary of 2 and placed one after another. Dynamically allocated memory refers to memory blocks allocated on a program's heap at runtime. For the purpose of this presentation it is

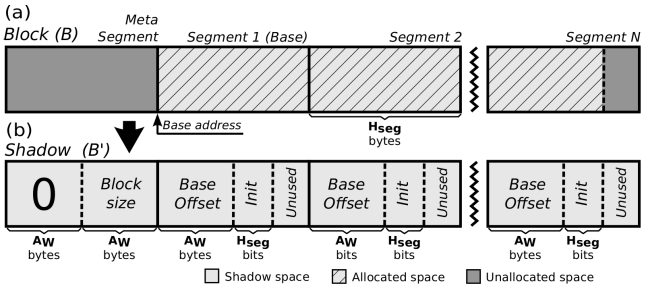

Figure 1. **(a)** Segment-based representation of an allocated memory block B, and **(b)** its shadow representation, where A_W denotes the byte-width of a memory address and H_{seg} is the byte-length of an application segment.

assumed that dynamically allocated memory blocks are aligned at some boundary A_{bnd} and padded with at least $2{\times}A_W$ bytes.

3. Segment-based Shadow State Encoding

This section introduces *segment-based* shadow state encoding applicable for aligned allocation of memory blocks.

3.1 Segment-Based Representation of Memory Blocks

Any memory block can be represented using equal-length segments (i.e., contiguous memory regions) as shown in Figure 1(a). Let $H_{seg} \geq 0$ denote the byte-length of a segment. A memory block B of L bytes in length is represented by a *meta-segment* and $N = \lceil L/H_{seg} \rceil$ *block segments*, where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. Block segments are numbered $1, 2, \ldots, N$ from left to right. The base address of the i^{th} segment is given via $base_B + (i-1) \times H_{seg}$. The first block segment is called the *base segment*. It is preceded by the meta-segment whose base address is $a - H_{seg}$.

The meta-segment and, for the case when L is not divisible by H_{seg}, the $H_{seg} - (L \bmod H_{seg})$ trailing bytes of the last segment, belong to unallocated space. Memory overhead of segment-based representation of a memory block B using H_{seg}-byte segments is thus H_{seg} if L is divisible by H_{seg}, and $2 \times H_{seg} - (L \bmod H_{seg})$ bytes otherwise.

A concrete example of a segment-based representation of an allocated memory block B of 40 bytes with base address 0x100 using 16-byte segments is shown via Figure 2(a). Address intervals $[0\text{xF0}, 0\text{xFF}]$ (meta-segment) and $[0\text{x128}, 0\text{x12F}]$ are unallocated.

3.2 Shadow Memory Structure

This section details the shadow state encoding. The general scheme is given in Figure 1 and a concrete example in Figure 2. We assume that an application block B is tracked by a shadow block B', such that for each segment in B there is corresponding segment in B'. Segments in B' can be scaled. Let H_{seg} and S_{seg} denote the lengths of application and shadow segments respectively.

Assumptions. Segment-based shadow state encoding requires that application memory blocks be aligned at a boundary A_{bnd} divisible by an application segment length H_{seg}. The purpose of this requirement is twofold. Firstly, it guarantees that application segments do not overlap. Secondly, it makes it possible to find a segment base address using modulo operation. Indeed, in this case for an address $addr$ belonging to a segment of B, the base address of the segment is $addr - (addr \bmod H_{seg})$.

The proposed encoding also requires every application block be to padded with at least H_{seg} bytes to reserve space for a meta-segment. It should be noted that in practice such requirements (i.e., aligned allocation and padding) do not present an issue.

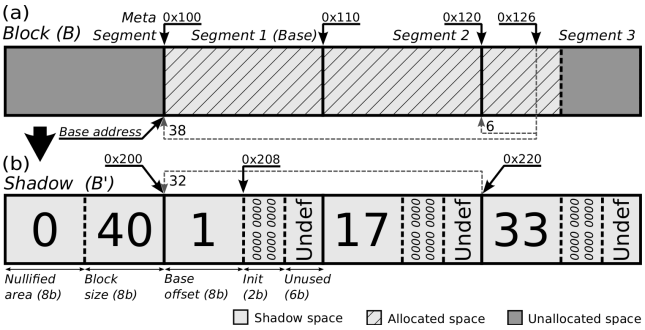

Figure 2. **(a)** A segment-based representation of a memory block B of length L=40 bytes and base address 0x100 by segments of size H_{seg}=16 bytes, and **(b)** its shadow representation with address-width A_W=8 bytes and compression ratio 1:1 (i.e. S_{seg}=H_{seg}). "Undef" marks unused regions.

We also assume that the length of a shadow segment S_{seg} cannot be less than $2{\times}A_W$ bytes. Shadow segments that are not a part of the shadow representation of an allocated block are zeroed out.

There is no length-related limitation for application memory segments, but there is a correspondence between lengths of application and shadow segments and shadow compression ratio. Shadow compression ratio is defined by H_{seg}:S_{seg}. That is, longer application segments lead to more compact shadow memory. However, due to per-block memory overhead of a segment-based representation (cf. Section 3.1), an increase in an application segment length also increases memory overhead. For instance, in a 32-bit system 8-byte application segments allow for shadowing of 1 application byte by 1 shadow byte. Allocation of a memory block of L bytes then additionally requires 8 bytes if L is divisible by 8, and $16 - (L \bmod 8)$ bytes otherwise. Application segments of 32 bytes allow to reduce the amount of shadow memory by the factor of 4, however per-block overhead then increases to 32 bytes for blocks whose lengths are divisible by 32, and to $64 - (L \bmod 32)$ bytes otherwise.

Meta-Segment Encoding. The meta-segment M' of the shadow block B' tracking B is encoded as follows (cf. Figure 1):

- The lower A_W bytes of M' are zeroed-out (to indicate an unallocated segment).
- The A_W bytes of M' following its lower A_W bytes store the byte-length of B.
- The remaining bytes of M' (if any) are unused.

Block-Segment Encoding. A block-segment S' of the shadow block B' tracking B is used as follows (cf. Figure 1):

- The lower A_W bytes of S' capture the offset from the base address of B' to the base address of S' incremented by one. Thus, a non-zero value indicates that at least the first byte of the corresponding application segment is allocated.
- The H_{seg} bits following the lower A_W bytes of S' capture per-byte initialization of the corresponding segment from B. The i^{th} bit in the shadow segment following its first A_W bytes is set to 1 whenever the i^{th} byte in the corresponding application memory segment is initialized.
- The remaining higher bytes of S' (if any) are unused.

Encoding Example. Figure 2 illustrates segment-based shadow state encoding of a 40-byte application memory block B tracked via shadow block B'. This example assumes 64-bit architecture, shadow compression ratio of 1:1, 16-byte segments and an alignment bound of 16.

Both B and B' are represented by a meta-segment and three 16-byte block segments. The lowest 8 bytes of the shadow meta-segment are nullified (indicating an unallocated segment) and its highest 8 bytes store the byte-length of B. Further, the lowest 8 bytes of each block segment in B' store a byte offset from the base address of this segment to the base address of the shadow block incremented by one. The following 16 bits store per-byte initialization status of the corresponding application segment. The example assumes that B is not initialized. Finally, the remaining 6 bytes of each block segment in B' are unused.

3.3 Computing Block-level Properties of an Address

Let $addr$ be an address belonging to an application memory block shadowed as described in Section 3.2. One of the key features of a segment-based representation is that the base address of the segment $addr$ belongs to can be computed by modulo operation. One can thus use the offset stored by the corresponding shadow segment to compute the base address ($base_{addr}$) of the memory block $addr$ belongs to and retrieve the byte-length of that block stored by the shadow meta-segment. Finally, one can subtract $base_{addr}$ from $addr$ to arrive to its byte offset. This section further gives detailed explanations on how to use the segment-based shadow state encoding to compute block-level properties of a memory address.

Let $ReadNum(a)$ denote retrieving a number stored in the A_W bytes starting at some memory address a,

$ReadBit(a, i)$ denote retrieving 0 or 1 value stored in the i^{th} bit past some memory address a,

$Shadow(a)$ be a mapping translating the base address a of an application segment into the base address of the corresponding shadow segment,

$Uspace(s)$ be a mapping[1] translating base address s of a shadow segment into the base address of the corresponding application segment.

The length of the memory block $addr$ belongs to, the base address of that block, the byte offset of $addr$ and its initialization status can be computed as follows (cf. Figure 3). Offset of $addr$ relative to the base address of its segment:

$$segoff_{addr} = addr \bmod H_{seg} \tag{1}$$

Base address of the segment $addr$ belongs to:

$$seg_{addr} = addr - segoff_{addr} \tag{2}$$

Base address of the shadow segment seg_{addr}:

$$seg_{sh} = Shadow(seg_{addr}) \tag{3}$$

Offset from the base address of the shadow segment to the base address of the shadow block:

$$baseoff_{sh} = ReadNum(seg_{sh}) - 1 \tag{4}$$

Base address of the shadow block seg_{sh} belongs to:

$$base_{sh} = seg_{sh} - baseoff_{sh} \tag{5}$$

Length of the memory block $addr$ belongs to:

$$Length = ReadNum(base_{sh} - S_{seg} + A_W) \tag{6}$$

Base address of the memory block $addr$ belongs to

$$base_{addr} = Uspace(base_{sh}) \tag{7}$$

Byte offset of $addr$ within its block:

$$baseoff_{addr} = addr - base_{addr} \tag{8}$$

[1] $Shadow$ and $Uspace$ can be seen as inverse mappings of the form $a \mapsto a \times Scale + Offset$ with suitable values of $Scale$ and $Offset$.

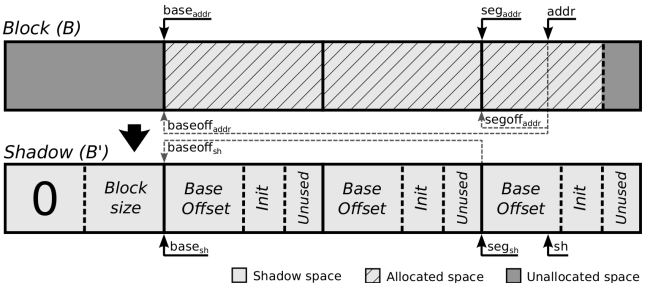

Figure 3. Computing block-level properties using segment-based shadow state encoding of allocated memory (Section 3.3).

An application memory address $addr$ is identified as belonging to a program's allocation if:

$$ReadNum(seg_{sh}) \neq 0 \text{ and } 0 \leq baseoff_{addr} < Length \tag{9}$$

It should be noted that $addr$ can belong to a partially allocated segment. Thus, to detect whether $addr$ is allocated one first needs to identify whether $addr$ lies within a segment which belongs to an allocated block, and if so, further check that the byte offset of $addr$ is not greater than the length of this block (cf. (9)).

An application memory address $addr$ is identified as initialized by a previous write if:

$$\begin{gathered} addr \text{ belongs to a program's allocation, and} \\ ReadBit(seg_{sh} + A_W, segoff_{addr}) = 1 \end{gathered} \tag{10}$$

3.4 Numeric Example

This section now presents a concrete example of computations discussed in Section 3.3 using example of Figure 2 which encodes a 40-byte application memory block B by a shadow block B'. Assume that B starts at address 0x100, while $Shadow(a) = a +$ 0x100, and $Uspace(sh) = sh -$ 0x100. Consequently B' starts at address 0x200.

Consider a memory location at address $addr$ of 0x126 (38 bytes past the base address of B). Applying Equations (1)–(8), one can compute the length of the memory block $addr$ belongs to, the base address of that block and the byte offset of 0x126 as follows:

$$\begin{gathered} segoff_{addr} = \text{0x126} \bmod 16 = 6, \\ seg_{addr} = \text{0x126} - 6 = \text{0x120}, \\ seg_{sh} = Shadow(\text{0x120}) = \text{0x220}, \\ baseoff_{sh} = ReadNum(\text{0x220}) - 1 = 33 - 1 = 32, \\ base_{sh} = \text{0x220} - 32 = \text{0x200}, \\ Length = ReadNum(\text{0x200} - 16 + 8) = 40, \\ base_{addr} = Uspace(\text{0x200}) = \text{0x100}, \\ baseoff_{addr} = \text{0x126} - \text{0x100} = \text{0x26}. \end{gathered}$$

Thus, by (9), 0x126 is identified as belonging to a program's allocation. Finally, one can compute initialization status of 0x126 using (10). Since 0x126 is the 7^{th} byte in its application segment (i.e., at offset 6), its initialization is tracked by the 7^{th} bit past $seg_{sh} + A_W =$ 0x228.

Consider now the address 0x12A. Applying Equations (1)–(9), one can establish that it belongs to a partially allocated segment, but lies past the end of B and therefore does not belong to allocation.

4. Offset-based Shadow State Encoding

Unlike heap allocation, blocks allocated on a program's stack are typically not aligned. Since stack blocks are often small, introducing alignment sufficient to apply the segment-based shadow state encoding discussed in the previous section is likely to introduce

Code	*(Len, Off)*	*Code*	*(Len, Off)*	*Code*	*(Len, Off)*
1	$(1,0)$	13	$(5,2)$	25	$(7,3)$
2	$(2,0)$	14	$(5,3)$	26	$(7,4)$
3	$(2,1)$	15	$(5,4)$	27	$(7,5)$
4	$(3,0)$	16	$(6,0)$	28	$(7,6)$
5	$(3,1)$	17	$(6,1)$	29	$(8,0)$
6	$(3,2)$	18	$(6,2)$	30	$(8,1)$
7	$(4,0)$	19	$(6,3)$	31	$(8,2)$
8	$(4,1)$	20	$(6,4)$	32	$(8,3)$
9	$(4,2)$	21	$(6,5)$	33	$(8,4)$
10	$(4,3)$	22	$(7,0)$	34	$(8,5)$
11	$(5,0)$	23	$(7,1)$	35	$(8,6)$
12	$(5,1)$	24	$(7,2)$	36	$(8,7)$

Table 1. Mapping of primary shadow codes to block lengths and byte offsets.

significant memory overhead. This section describes an alternative *offset-based* encoding scheme for tracking unaligned memory blocks.

4.1 Shadow Memory Structure

In the offset-based encoding application memory is shadowed via two shadow spaces both of which shadow one byte of application memory by one byte of shadow memory. This paper refers to these shadow spaces as to *primary* and *secondary* shadows.

4.1.1 Primary Shadow

Each byte of an application memory block B is tracked by a primary shadow byte of the following structure.

The 6 lower bits of a primary shadow byte store *shadow status code* of the application byte it tracks. Shadow status codes are given by numbers ranging from 0 to 63, such that the shadow status code sh_{stat} tracking application address $addr$ has the following interpretation:

- if the value of sh_{stat} is 0, then $addr$ is unallocated.
- if the value of sh_{stat} is between 1 and 36, then $addr$ belongs to an allocated memory block whose length is less than or equal to 8 bytes. In this case the value of sh_{stat} encodes the length of B and the byte offset of $addr$ using the mapping of Table 1.
- if the value of sh_{stat} is between 48 and 63, then $addr$ belongs to an allocated memory block whose length is greater than 8 bytes and the value of sh_{stat} decremented by 48 indicates an offset relative to a location in the secondary shadow where the length and the byte offset of $addr$ are stored (as further explained in Section 4.1.2).
- Status code values ranging from 37 to 47 are unused.

The 7^{th} highest bit of a primary shadow byte stores initialization status of the application byte it shadows, where 1 denotes an initialized byte and 0 denotes uninitialized. The 8^{th} highest bit of a primary shadow byte is unused.

Encoding Example. Figure 4 illustrates segment-based encoding for an allocated byte and a 4-byte memory block B. Since B is less than 9 bytes, it has no secondary shadow representation.

4.1.2 Secondary Shadow

The secondary shadow tracks offsets and lengths of application blocks whose lengths are greater than 8 bytes. Application blocks whose lengths are less than or equal to 8 bytes are not represented in the secondary shadow. An application block of L bytes ($L > 8$) is tracked by an equal size block in the secondary shadow divided into 8-byte segments such that:

- the 4 lower bytes of each secondary shadow segment capture the length of the shadowed block, and
- the 4 higher bytes of each secondary shadow segment store an offset from the base address of the segment to the base address of the secondary shadow block.

The trailing $L \bmod 8$ bytes of the secondary shadow block mapped to an application-space block are unused.

Encoding Example. Figure 5 illustrates segment-based shadow state encoding for an 18-byte memory block B with base address 0x100, tracked by primary shadow block B' and a secondary shadow block B''. For the purpose of this example, access to primary and secondary shadow regions is enabled using displacement of an address by offsets 0x100 and 0x200 respectively.

Secondary shadow block B'' is represented by two 8-byte segments and two unused trailing bytes. 4 lower bytes of each segment in B'' store the length of B and 4 higher bytes store an offset from the base address of a segment to the base address of B''.

Each byte in the primary shadow block B' stores a shadow status code capturing an offset to the base address of the nearest shadow segment. That is, using an offset in the primary shadow one can access a location in the secondary shadow to determine the length of the tracked block.

4.2 Computing Block-level Properties of an Address

Let $addr$ be an application address belonging to an allocated memory block shadowed using offset-based shadow state encoding. This section explains how to compute its block-level properties.

Let $Shadow_{prim}(a)$ be a mapping translating an application address a into a corresponding address in the primary shadow,

$Shadow_{sec}(a)$ be a mapping translating an application address a into a corresponding address in the secondary shadow,

$Uspace_{prim}(s)$ be a mapping translating a primary shadow address s into a corresponding application-space address,

$Uspace_{sec}(s)$ be a mapping translating a secondary shadow address s into a corresponding application-space address,

$ReadStat(a)$ denote reading a number stored in the 6 lower bits of a memory location given by address a, and

$ReadInt(a)$ denote reading a number stored in the 4 bytes starting at address a.

The length of the memory block $addr$ belongs to, the base address of that block and the byte offset of $addr$ can be found using its shadow status code sh_{stat}. It is computed as the number stored in the 6 lower bits of the primary shadow byte mapped to $addr$:

$$sh_{stat} = ReadStat\ (Shadow_{prim}(addr))$$

Remaining computations depend on the value of sh_{stat}.

If the value of sh_{stat} is 0, then $addr$ is unallocated.

If the value of sh_{stat} is between 1 and 36, then the length of the memory block and the byte offset of $addr$ are found as shown in Table 1. The base address of the block $addr$ belongs to can be obtained by subtracting the byte offset of $addr$ from $addr$.

If the value of sh_{stat} is between 48 and 63, then properties are computed as follows:

Base address of a secondary shadow segment tracking $addr$:

$$sh_{sec} = Shadow_{sec}(addr) - (sh_{stat} - 48) \tag{11}$$

Length of the memory block $addr$ belongs to:

$$Length = ReadInt(sh_{sec}) \tag{12}$$

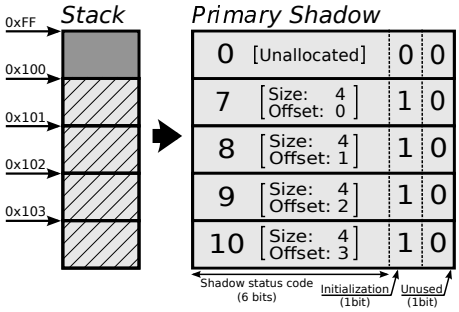

Figure 4. Offset-based encoding of an unallocated byte at address 0xFF followed by an allocated block of 4 bytes with base address 0x100.

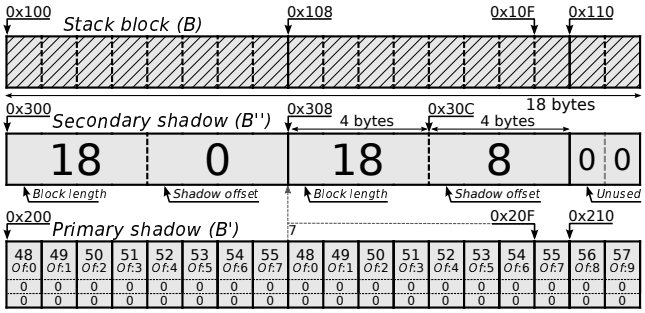

Figure 5. Offset-based encoding of an 18-byte memory block starting at address 0x100. Shadow status codes (48-57) in the primary shadow block (below) indicate offsets (0-9) used to find base addresses of nearest secondary shadow segments.

Base address of the memory block $addr$ belongs to:

$$base_{addr} = Uspace_{sec}(sh_{sec}) - ReadInt(sh_{sec} + 4) \quad (13)$$

Byte offset of $addr$ within its block:

$$baseoff_{addr} = addr - base_{addr} \quad (14)$$

4.3 Numeric Examples

This section now presents two concrete examples showing computations discussed in Section 4.2.

Shadowing a Short Block. Consider example encoding of a 4-byte block illustrated by Figure 4. The computations are directly based on the shadow status code and Table 1. For instance, for application address $addr = 0\text{x}102$, shadow status code 9 indicates that 0x102 belongs to a block of 4 bytes and its byte offset is 2. Thus, $base_{addr} = 0\text{x}102 - 2 = 0\text{x}100$. Finally, the initialization status of 0x102 can be determined via the 7^{th} highest bit (here, set to 1) of the primary shadow byte address 0x102 is mapped to.

Shadowing a Long Block. Consider example encoding discussed in Section 4.1.2 and illustrated by Figure 5, where an 18-byte block B is tracked via primary and secondary shadow blocks B' and B'' respectively. The example considers $Shadow_{prim}(a) = a + 0\text{x}100$, $Uspace_{prim}(sh) = sh - 0\text{x}100$, $Shadow_{sec}(a) = a + 0\text{x}200$, and $Uspace_{sec}(sh) = sh - 0\text{x}200$.

Consider application address $addr = 0\text{x}10\text{F}$. Shadow status code mapped to 0x10F in the primary shadow (i.e., at address 0x20F) is 55, which indicates that 0x10F belongs to a memory block whose length is greater than 8 bytes.

Applying (11) and (12), one can compute the base address of a secondary shadow segment storing B's length and read this length:

$$sh_{sec} = Shadow_{sec}(0\text{x}10\text{F}) - (55 - 48) = 0\text{x}308,$$
$$Length = ReadInt(0\text{x}308) = 18$$

The offset from the base address of the secondary shadow segment to the base address of the secondary shadow block B'' is stored in the 4 higher bytes of the segment (i.e., $ReadInt(0\text{x}308 + 4)$). The base address of B is then obtained by using the offset to compute the base address of the secondary shadow block and mapping it back to an application address using (13):

$$base_{addr} = Uspace_{sec}(0\text{x}308) - ReadInt(0\text{x}308 + 4)$$
$$= 0\text{x}108 - 8 = 0\text{x}100$$

Finally, the byte offset of 0x10F is found by applying (14):

$$baseoff_{addr} = 0\text{x}10\text{F} - 0\text{x}100 = 15$$

5. Implementation

Segment- and offset-based shadow state encodings detailed in Sections 3 and 4 have been implemented in the Frama-C plug-in called E-ACSL [9]. Frama-C [21] is an open-source framework for analysis of C programs. E-ACSL plug-in is a runtime verification tool that accepts a C program P annotated with formal specifications written in the E-ACSL specification language and generates a new program P' that fails at runtime whenever an annotation is violated. If for a given program execution every annotation is satisfied, P' is functionally equivalent to P. Otherwise said, E-ACSL inserts inline monitors generated from formal specifications. These specifications can either be provided by the end-user, or automatically generated by another tool. For instance, RTE plug-in [1] of Frama-C can automatically generate such annotations for most undefined behaviors (e.g., memory violations and arithmetic overflows). Monitors generated by E-ACSL plug-in are able to track them automatically. Among others, formal specifications include memory-related annotations such as `\valid(p)`, `\initialized(p)`, `\base_addr(p)`, `\block_length(p)`, and `\offset(p)`. For a given pointer p, they respectively denote the validity of p, whether the value pointed to by p has been initialized, the base address of the memory block p belongs to, byte-length of that block and the byte offset of p relative to the base address.

Predicates of the E-ACSL specification language can be used to detect out-of-bounds violations. For instance, for a dereference `*(p+i)` the E-ACSL expression `\base_addr(p) == \base_addr(p+i)` specifies that `p+i` belongs to the same memory block that `p` points to. This detects an out-of-bounds access if both `p` and `p+i` point to two different allocated areas. The E-ACSL specification language can express a large variety of contract properties (see [9]) that go far beyond out-of-bounds checks, where block-level properties can be also very useful. For instance, for a C standard library call `q = strchr(str,c)` returning a pointer to the first occurrence of character `c` in the string `str`, one can specify that `q` belongs to the same memory block as `str` but with a greater offset:

```
\base_addr(str) == \base_addr(q) && \offset(str) <= \offset(q),
```

and that `c` does not appear in string `str` before `q`:

```
\forall integer i, 0 <= i < \offset(q)-\offset(str)
                ==> str[i] != c.
```

In order to support E-ACSL memory predicates, code generated by E-ACSL relies on a C runtime memory library (RTL): memory modifications of the program are stored in the RTL by the monitor, while checking annotations corresponds to querying it. Before program instrumentation, static analysis is performed to safely remove unnecessary memory instrumentation to improve efficiency of the monitor [18]. RTL was initially implemented using a Patricia trie (a compact prefix tree) [22] that stores block metadata in its leaves and uses base addresses of memory blocks as keys associated with trie nodes. Routing from the root to a leaf is ensured by internal

nodes, each of which contains the greatest common prefix of base addresses stored in its successors. The Patricia trie was replaced by an implementation of memory shadowing described in Sections 3 and 4 without modifying the E-ACSL code generator.

The current implementation of RTL uses segment-based encoding to shadow memory on a program's heap, while offset-based encoding has been adopted for tracking of automatic and static allocations. To enforce alignment and padding required by the segment-based encoding, a customized jemalloc [12] memory allocator has been used. The present implementation supports both 32-bit and 64-bit architectures. The former case has 4-byte address-width and uses 8-byte segments and an alignment bound of 8, while the latter has 8-byte address-width and uses 16-byte segments and an alignment bound of 16. In both cases compression ratio is 1:1.

Shadow memory implemented by the present RTL is based on a direct mapping scheme and represented by 5 disjoint regions: a shadow space tracking the heap, and primary and secondary shadow spaces tracking stack and global memory. Switching between different shadow spaces is enabled by range checking. At runtime an address is first identified as belonging to stack, heap or global memory by testing its value against recorded address ranges of respective memory segments, then, an appropriate encoding is selected to compute properties of the address.

Practical application of the segment- and offset-based encodings is investigated in the following section that reports on experimentation of the present E-ACSL implementation using SPEC CPU benchmarks [35].

6. Experimental Results

Experiments presented in this section focus on runtime and memory overheads incurred by E-ACSL implementation of segment- and offset-based shadow state encodings for a problem of spatial memory safety in C programs.

6.1 Objectives and Experimental Setup

The main research question addressed by the experimentation is to identify overhead costs of memory tracking using segment- and offset-based shadow state encodings with respect to runtime overheads incurred by state-of-the-art monitoring techniques for a problem of checking validity of pointer and array accesses.

The above research question is investigated using an experiment that compares runtime and memory overheads of an implementation of the present approach incurred during monitoring of programs selected from SPEC CPU datasets with overheads of AddressSanitizer [31], Dr. Memory [6], MemCheck [33] and patricia trie implementation of E-ACSL [22]. For brevity, E-ACSL implementations using shadow memory and patricia trie are referred to E-ACSL-Shadow and E-ACSL-Trie respectively.

During the present experimentation a series of executions of original and instrumented (or monitored) programs were performed and their runtimes and memory consumption measured. A runtime of a program accounts for real time between the program's invocation and its termination, whereas memory consumption indicates maximum resident set size (RSS) of the process during its lifetime. Further, runtime and memory overheads of instrumented (or monitored) programs relative to the execution time and memory consumption of uninstrumented programs were calculated. To account for variance due to external factors, such as test automation process or system I/O, the overheads were calculated using an arithmetic mean over 10 runs of the modified and the original executables.

Programs monitored using E-ACSL-Shadow and E-ACSL-Trie were instrumented using E-ACSL annotations validating safety of pointer or array accesses. The annotations were generated automatically by the RTE plugin of Frama-C (cf. Section 5). For runs of AddressSanitizer, Dr. Memory and MemCheck, which make use of built-in analyses, no external specifications were provided. Original and E-ACSL-instrumented programs were compiled using GCC.

	E-ACSL	ASAN	MCHECK	DRMEM 32/64
Heap Tracking	+	+	+	+ +
Stack Tracking	+	+	−	+ +
Global Tracking	+	+	−	+ +
Allocation	+	+	+	+ +
Initialization	±	−	+	+ −
Pointer Init	+	−	−	− −
Bounds Check	+	±	±	± ±
Arith. Overflow	+	−	−	− −
Read-only	+	−	−	− −
Block Properties	+	−	−	− −
Heap Leak	±	+	+	+ +

Table 2. Properties tracked during experimentation.

AddressSanitizer executables were generated by its GCC branch version 5.4.0. Runs of MemCheck (version 3.10.1) and Dr. Memory (version 1.11.0-2) were performed using executables compiled from uninstrumented programs. Since current version of Dr. Memory does not support initialization checks in 64-bit executables (as reported by the tool), for evaluation of Dr. Memory both 64- and 32-bit binaries were used. 32-bit binaries were compiled on a 64-bit architecture using `-m32` GCC flag. Overhead ratios of 32-bit executables under Dr. Memory monitoring were computed using uninstrumented runtimes of the same executables.

Table 2 summarizes properties tracked by the tools during the present experimentation. Since different implementations of E-ACSL are equivalent with respect to the checked properties it is shown as a single instance. Here, + indicates that a feature is fully supported, − denotes unsupported functionality and ± stands for partial support. All tools used during the experimentation enable tracking of heap memory. E-ACSL, AddressSanitizer and Dr. Memory also monitor stack and global memory, whereas MemCheck does not. MemCheck enables checking of load and store instructions to detect use of unallocated memory and initialization errors. Dr. Memory enables similar checks, however does not presently support initialization checks in 64-bit executables. E-ACSL and AddressSanitizer check pointer dereferences and array subscripts for allocation errors, but do not check initialization of every read. It should be noted that even though E-ACSL supports arbitrary initialization checks this functionality has been disabled during experimentation. Instead, E-ACSL checked every dereferenced pointer to be initialized (denoted *Ptr Init*), while the rest of the tools did not. For detection of out-of-bounds AddressSanitizer, Dr. Memory and MemCheck use redzoning, and E-ACSL tracks precise bounds of allocated blocks. Since redzoning tries to detect out-of-bounds accesses using gaps between allocated block such an approach is unsound (indicated via ± in Table 2). Unlike the rest of the tools E-ACSL also checks arithmetic overflows in array subscripts, tracks read-only memory and computes block-level properties of memory addresses. Finally, all tools detect memory leaks. The present implementation of E-ACSL leak detection does not support tracking locations of leaked blocks.

This experimentation uses 17 C programs selected from SPEC CPU 2000 and 2006 datasets ranging from 74 to 36,037 (on average, 9,345) lines of code. Remaining C programs were rejected due to current limitations of E-ACSL instrumentation engine independent of the proposed encodings. Runs of SPEC programs were performed using inputs provided by the training input dataset of SPEC. The platform for all results reported here was 2.30GHz Intel i7 processor with 16Gb RAM, running Gentoo Linux. Time and memory consumption measurements were taken using GNU `time` tool.

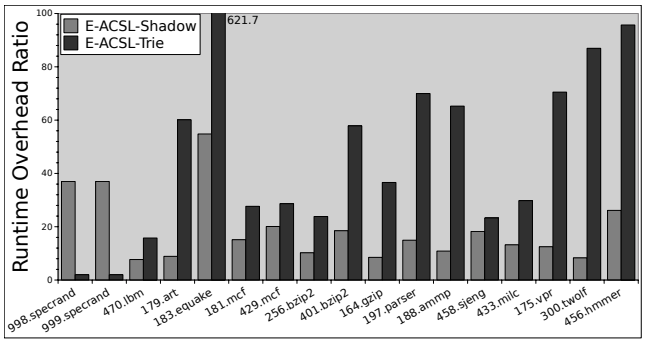

Figure 6. Runtime overheads of SPEC CPU programs for E-ACSL-Shadow and E-ACSL-Trie.

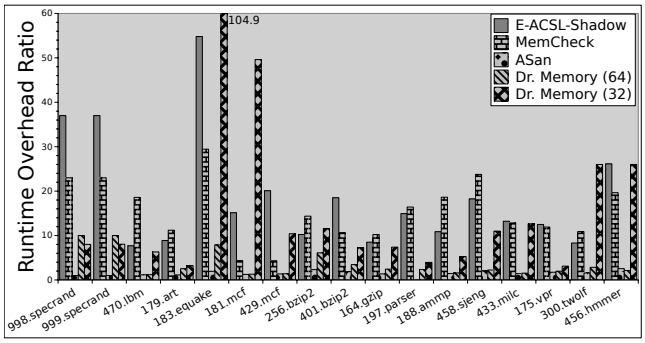

Figure 7. Runtime overheads of SPEC CPU programs for AddressSanitizer, MemCheck, Dr. Memory and E-ACSL-Shadow.

6.2 Runtime Overheads

Figure 6 shows runtime overheads of E-ACSL-Shadow and E-ACSL-Trie relative to execution time of unobserved runs. Runtime overheads of E-ACSL-Shadow are approximately 19 times the normal execution with the highest spike of approximately 55 times in `181.mcf` and the lowest overhead of approximately 8 times in `470.lbm`. Runtime overheads of E-ACSL-Trie, ranging from 2 to approximately 662 times, have a mean of 77.5 times.

On average the runtime overheads of E-ACSL-Shadow are significantly lower than those of E-ACSL-Trie. One reason for such behaviour is that updating shadow memory is less expensive than trie lookups. Further, runtime overheads of E-ACSL-Trie increase proportionally to the number of tracked memory blocks. An increase in the number of tracked blocks results in an increase of a trie height that consequently contributes to the cost of a lookup. This is well demonstrated by over 600 times overhead in `183.equake` and over 95 times overhead `456.hmmer`, programs which allocate a large number of memory blocks. Note that in the runs of `999.specrand` and `998.specrand` programs the runtime overheads of E-ACSL-Trie are lower than those of E-ACSL-Shadow. This is because these programs are small (with execution time under 0.1 sec.) and E-ACSL-Shadow is penalized by the time required to initialize its shadow space.

Figure 7 shows runtime overheads of E-ACSL-Shadow, AddressSanitizer, MemCheck and Dr. Memory relative to the normalized execution time of unobserved runs. Overheads of MemCheck, ranging from approximately 4.3 to 29.5 times the unobserved execution average to 15.49 times. The overheads AddressSanitizer are significantly lower, averaging to 1.58 times with the maximal overhead of 2.55 times in `456.hmmer`. In runs of 64-bit executables

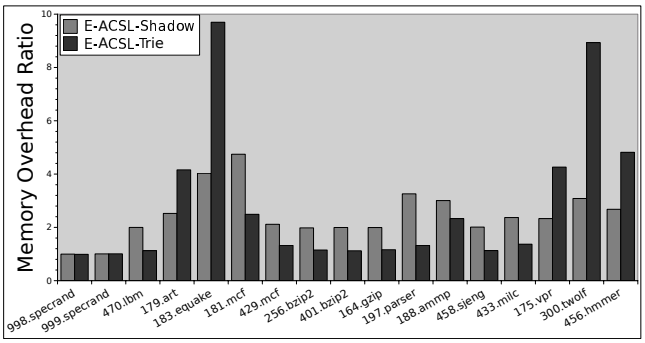

Figure 8. Memory overheads of SPEC CPU programs for E-ACSL-Shadow and E-ACSL-Trie.

Dr. Memory has an average overhead of 3.59 times, a minimum of 1.19 and a maximum of 10 times. During monitoring of 32-bit binaries, however, this tool incurs significantly higher overheads with an average slow-down factor of 18 times and a highest spike of approximately 105 times in `181.equake`. Such a difference in runtime overheads between 32- and 64-bit programs can be explained by unsupported initialization checks in 64-bit binaries.

It can be seen that runtime overheads of shadow techniques depend on encoding of shadow values and the number of runtime checks. The issue is demonstrated by the difference in overheads of MemCheck, which checks per-bit initialization and AddressSanitizer that monitors allocation of 8-byte sequences by 3 bits and does not detect initialization errors. Further, an increase in overhead of Dr. Memory during its 32-bit tracking (comparing to a 64-bit one) can be explained by additional initialization checks.

The experimentation shows that the runtime overheads of E-ACSL-Shadow are close (but on average higher) to those of 32-bit tracking MemCheck and Dr. Memory. Notably, block-level analysis of E-ACSL-Shadow brings the benefit of detecting a wider range of issues. For instance, E-ACSL-Shadow has detected an out-of-bounds stack access in `175.vpr`, missed by the other tools. The issue discovered relates to using a stack buffer after the end of the scope of its definition has been reached. For fairness of the evaluation, when detecting such a violation the default behaviour of E-ACSL-Shadow was modified in order to report it but continue the execution of the program instead of stopping it. Further, even though the overheads of AddressSanitizer and 64-bit tracking of Dr. Memory are considerably lower, these tools do not track block properties and cannot identify memory violations involving accesses to uninitialized memory.

In summary, the results of this experimentation suggest that segment- and offset-based encodings solve the performance bottleneck of E-ACSL-Trie while being still capable of tracking memory at a block-level. Further, overhead results of E-ACSL-Shadow (avg. 19 times), that are close to MemCheck (avg. 15.49 times) and 32-bit tracking of Dr. Memory (avg. 18 times), indicate that the proposed technique can be used as a replacement for these tools adding the benefit of tracking memory at a block-level and detecting a wider range of runtime errors. Finally, the experimentation has shown that the overheads of E-ACSL-Shadow are still significantly higher that those of AddressSanitizer. This extra overhead, however, accounts for additional features such as tracking block boundaries.

6.3 Memory Overheads

Figure 8 shows memory consumption overheads of E-ACSL-Shadow and E-ACSL-Trie relative to memory consumption during executions of uninstrumented programs. On average, memory overheads of E-ACSL-Shadow are 2.48 times the unobserved ex-

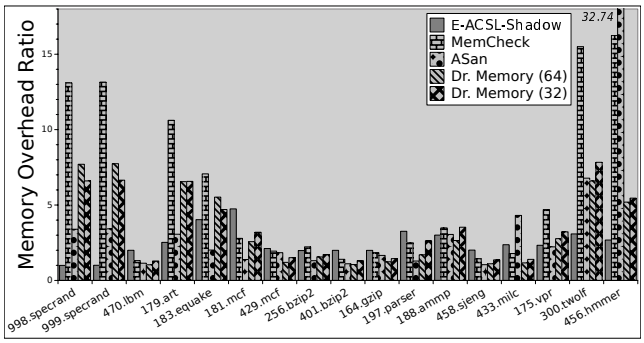

Figure 9. Memory overheads of SPEC CPU programs for AddressSanitizer, MemCheck, Dr. Memory and E-ACSL-Shadow.

ecution with the maximum and minimal results of 1.01 and 4.74 times respectively. E-ACSL-Trie uses more memory, having the mean of 2.85 times, the maximum of 9.7 and the minimum of 1.01 times compared to memory consumption of unobserved programs. The results indicate that on average E-ACSL-Trie is less space efficient than E-ACSL-Shadow. Such a result in unexpected because E-ACSL-Trie uses a fixed amount of memory to represent the bounds of a memory block (48 bytes in the present experimentation), whereas memory overheads of E-ACSL-Shadow, increase proportionally to the overall amount of allocated memory. E-ACSL-Trie, however, is penalised in the runs where many small stack blocks are used, where fixed-size representation of E-ACSL-Trie requires more memory to track bounds of a block than offset-based encoding. For instance, to represent a 4-byte integer offset-based encoding requires uses 4 bytes, whereas E-ACSL-Trie needs 12 times that. This issue is demonstrated by 9.7 times space overheads in `183.equake` and almost 9 times in `300.twolf`.

Figure 9 shows memory overheads of E-ACSL-Shadow, AddressSanitizer, MemCheck and Dr. Memory relative to the normalized memory consumption time of unobserved runs. On average the memory overheads of E-ACSL-Shadow (2.48 times) are the lowest of all tools, whereas MemCheck used more memory than the rest. The overheads of MemCheck, ranging from approximately 1.33 to 16.25 times the unobserved execution average to 5.95 times. The overheads AddressSanitizer are lower and average to 4.22 times with the minimal overhead of 1.04 times in `458.sjeng` and the maximal overhead of 32.74 times in `456.hmmer`. It should be noted that the average memory overhead of 4.22 times in runs of programs under AddressSanitizer instrumentation is also due to a memory consumption spike during a run of `456.hmmer`. This spike can be explained by the use of so-called quarantine-zones, where the heap allocator tries to returns new addresses rather than reuse freed memory. In most runs the overheads of AddressSanitizer are lower. Without taking this result into account the average memory overhead ratio of AddressSanitizer drops to 2.44 times compared to memory consumption of unobserved programs. Notably, this result is similar to the overhead incurred by E-ACSL-Shadow (2.48 times). On average, Dr. Memory is more space efficient then AddressSanitizer in runs of both 32- and 64-bit executables. 64-bit tracking incurs an average overhead of 3.37 times, a minimum of 1.04 in `470.lbm` and a maximum of 7.74 times in the run of `999.specrand`. Monitoring 32-bit programs requires slightly more memory: 3.55 times on average, with the highest spike of approximately 7.83 times in a run of `300.twolf` and the lowest result of 1.29 times in `470.lbm`.

In summary, the results of the experiment measuring memory consumption overheads indicate that additional functionality of the proposed shadow state encoding capable of tracking block bounds compare favourably to shadow-based techniques that track fewer properties at runtime. One of the reasons for such behaviour is that the present technique does not use redzoning when monitoring stack memory. The offset-based approach used to track stack memory does not need to introduce additional gaps between the allocated stack blocks to detect bounds violations. Further, even though segment-based encoding uses a 1:1 compression ratio in the virtual memory space of an executing process, $\frac{2}{3}$ of this memory is never used. An important factor contributing to higher overheads of AddressSanitizer, Dr. Memory and MemCheck is that they also use heap quarantine zones attempting to allocate memory at new addresses rather than reuse freed memory immediately. The implementation of the presented encoding relies on a different allocation strategy which makes use of freed memory and therefore is able to keep more compact representation of heap memory.

6.4 Threats to Validity

One of the issues that may have skewed the validity of the obtained results is the choice of programs and input data used during this experimentation. Even though this experimentation used programs and inputs that are well suited for estimating runtime overheads of memory monitoring, different programs or input values may lead to different overhead results. Further, due to technical issues with source code analysis this experimentation excluded several SPEC CPU programs from the experiment. Using those programs may have also affected the obtained overhead results.

Another issue refers to an instantiation of the segment-based shadow state encoding. E-ACSL-Shadow uses 16-byte segments and a 1:1 compression ratio. Using a different segment size or compression ratio may have affected overhead results. Further, E-ACSL-Shadow uses a direct mapping scheme, where a program's heap is managed though a single shadow space and stack and global memory is tracked via two disjoint (primary and secondary) shadow spaces each. A different implementation, for instance via a segmented shadow mapping scheme may have also lead to different overhead results. Comparing such different implementations is future work.

7. Related Work

This section now reviews related work focussing on techniques tracking memory state of an executing program at runtime for the purpose of detecting memory safety errors.

One of the oldest dynamic memory analysers (often referred to as memory debuggers) is Rational Purify [16]. Purify uses compile-time instrumentations to add additional instructions directly into object files. At runtime these instructions monitor memory accesses using shadow memory which tracks each byte of an application's memory by two bits denoting that byte's allocation and initialization. Memcheck [33], a more recent memory debugger built atop Valgrind instrumentation platform [27, 28], uses dynamic binary instrumentation (DBI) to instrument a program at runtime and shadow one byte of heap memory using 9 bits, such that one shadow bit is dedicated to storing allocation, while the remaining 8 bits track initialization with bit-precision. Even though MemCheck is one of the most popular tools for memory debugging (which is evident by its widespread use in large-scale projects [38]), this tool incurs overheads of 20-40 times the normal execution and tracks only heap memory, thus missing safety issues related to use of a program's stack or global variables. Such issues are addressed by SGCheck [34] – an experimental stack and global array overrun detector also built on top of Valgrind. Another notable memory debugger making heavy use of shadow memory is Dr. Memory [6] based on Dynamo RIO DBI platform [7]. Similar to Purify, Dr. Memory dedicates 2 shadow bits for tracking an application's byte.

Dr Memory incurs lower overheads than MemCheck, but tracks initialization with byte precision.

AddressSanitizer [31] is memory debugger targeting memory safety issues including out-of-bounds accesses and use-after-free violations. AddressSanitizer uses source-to-source transformations and utilizes a compact shadow state encoding allowing to track 8-byte sequences by 3 bits. Such compact representation allows for significant reduction of memory and runtime costs, leading to runtime overheads of approximately 2-2.5 times. AddressSanitizer, initially built on top of a clang compiler, has now been ported to gcc replacing mudflap [11]. MemorySanitizer [36] and ThreadSanitizer [32] are similar tools aiming detection of initialization errors and data races.

Unlike the above tools, which track individual bytes or bits, segment-based and offset-based encodings reported by this paper are also capable of tracking memory at a block-level without loss of byte-level information.

Tracking block bounds via shadow memory can potentially be enabled using generic metadata management techniques such as METAlloc [14]. Based on modern heap [13] and stack [23] organizations, METAlloc tracks pointers by splitting their addresses into offsets and page indices and further use them to access a metapage table storing pointers to per-block metadata. However, since METAlloc requires memory objects within their pages to share a non-trivial alignment, implementation of such a scheme for stack allocations is not straightforward and potentially leads to significant changes of a program's memory layout.

Similar to METAlloc, segment-base shadow state encoding also uses aligned allocation, but it does not require fixed alignment within individual pages. Further, offset-based shadow encoding allow to track stack blocks without any alignment, thus leaving stack memory layout intact. Finally, METAlloc suggests a generic scheme for tracking metadata and reports overheads of at most 20% for allocations and deallocations. Overheads of METAlloc for monitoring of memory safety errors, however, are not clear.

An alternative approach to tracking memory at runtime uses external databases. Jones and Kelly [20] enable runtime bounds checking using a splay tree whose nodes capture addresses of pointers, base addresses of their pointees and bound limits. At runtime operations on pointers (e.g., dereference or arithmetic) can be checked by using metadata accessible via tree lookups. CRED [30] improves on the work of Jones and Kelly by adding support for tracking out-of-bounds pointers using an additional auxiliary hash table. Dhurjati and Adve [10] also improve Jones and Kelly's bounds checking technique using a memory allocation scheme called Automatic Pool Allocation memory partitioning. Baggy bounds checking [3] is a similar technique using an allocator that constrains size and alignment of allocated blocks and uses array-based lookup. SoftBound [25] is also a related approach which tracks bounds for each pointer using a hash table or a large contiguous array.

Memory tracking can be enabled using *fat pointers*, a technique that extends pointer representation with extra information allowing to detect out-of-bounds violations. One of the techniques utilizing fat pointers is Cyclone [19] – a safe dialect of C designed to retain C semantics while preventing such issues as buffer overflows, format string vulnerabilities and memory management errors. Another user of fat pointers is [26] – a transformation system adding type safety guarantees to C programs. Kwon et al. [24] explores spatial safety using compact encoding of base and bound information encoded in a pointer.

Above techniques build upon source-code analysis, relations between program pointers and objects they point to aiming to detect when a pointer operates on a memory location outside of its bounds. This makes it difficult to use these approaches with binary instrumentation, where no pointer concept is available. Further, such techniques do not characterize each address as with shadow memory. In other words, given an arbitrary memory address outside of a pointer or object context these techniques may fail to identify precise bounds of the memory block it belongs to. Memory encodings presented in this paper are designed to answer such question using fast constant-time computations.

Techniques operating at a memory address-level have also been reported in the literature. Kosmatov et al. [22] use patricia trie [37] to store metadata about each memory block allocated by a program. One of the benefits of this approach is that it can use an address to compute metadata about a memory block the address belongs to. Typical metadata stored by this technique includes the block's base address, its length and per-byte initialization. One of the shortcomings of this approach is a costly trie computation that increase proportionally to allocation size. Jakobbsson et al. [17, 18] improves on this work by combining tracking via patricia trie with a shadow memory technique. Vorobyov et al. [40] have proposed a similar approach for a problem of memory leak detection using a red-black tree. Shadow encodings presented by this paper are also capable of tracking memory at a block level. However, segment-based and offset-based encodings benefit from fast constant time computations independent from the size of a program allocation.

8. Conclusion and Future Work

This paper presented segment-based and offset-based shadow state encoding schemes capable of tracking memory allocated by a program at runtime with byte- and block-level precision.

The presented shadow state encoding schemes have been implemented in the E-ACSL plug-in of Frama-C, allowing to detect block-level out-of-bound accesses not detected by MemCheck, Dr. Memory and AddressSanitizer memory debuggers. Empirical evaluation comparing performance overheads of these tools shows that the proposed shadow state encodings result in comparable runtime overheads over SPEC CPU programs and significantly outperform the initial implementation of E-ACSL based on Patricia trie. Further experimentation with SPEC CPU programs has shown that the average memory consumption of the proposed encodings compare favourably to the overheads incurred by the rest of the tools used in the evaluation.

There are several directions of future work available. Firstly, the authors are looking to adopt segment- and offset-based encodings for monitoring of different properties. Block-level tracking, capable of tracking program variables, can also be used for addressing information security vulnerabilities, for instance, during dynamic information flow analysis techniques tracking security levels of variables. Further, the presented encodings can potentially be used to address concurrency errors such as deadlocks or race conditions. Another potential direction of future work involves improvement of runtime overheads of the proposed technique. This paper reported on overhead results with 1:1 encoding for heap memory blocks for programs selected from SPEC datasets using a segment size of 16 bytes for a 64-bit system. To identify whether overheads can be improved by increasing or decreasing segment size more experimentation is required. The third direction relates to the use of offset- and segment-based encodings with E-ACSL. The present implementation already uses static analysis to reduce the amount of instrumentations without the loss of precision. Better static analyses can be used to further improve runtime overheads.

Acknowledgement. Part of the research work leading to these results has received funding for the S3P project from French DGE and BPIFrance. The authors thank the Frama-C team for providing tools and support. Many thanks to Prof. Richard E. Jones and the anonymous reviewers for many useful suggestions and advice.

References

[1] Runtime error annotation generation plug-in, 2009. `https://frama-c.com/rte.html`.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277. IEEE Computer Society, May 2008.

[3] 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 USENIX Security Symposium*, pages 51–66. USENIX Association, August 2009.

[4] A. Arya and C. Neckar. Fuzzing for security, April 2012. `http://blog.chromium.org/2012/04/fuzzing-for-security.html`.

[5] H. Bëck. Safer use of C code - running gentoo with AddressSanitizer, January 2016. `https://blog.hboeck.de/archives/879-Safer-use-of-C-code-running-Gentoo-with-Address-Sanitizer.html`.

[6] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[7] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[8] S. Christey. 2011 CWE/SANS top 25 most dangerous software errors. Technical Report 1.0.3, The MITRE Corporation, `http://www.mitre.org`, September 2011.

[9] M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1230–1235. ACM, March 2013.

[10] D. Dhurjati and V. S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the International Conference on Software Engineering*, pages 162–171. ACM, May 2006.

[11] F. C. Eigler. Mudflap: pointer use checking for C/C++. In *Proceedings of the GCC Developers Summit*, pages 57–70, May 2003.

[12] J. Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the Technical BSD Conference*, April 2006.

[13] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc, 2009. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[14] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos. METAlloc: Efficient and comprehensive metadata management for software security hardening. In *Proceedings of the European Workshop on System Security*, EuroSec '16, pages 5:1–5:6. ACM, 2016.

[15] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, 2012. ACM.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, January 1992.

[17] A. Jakobsson, N. Kosmatov, and J. Signoles. Fast as a shadow, expressive as a tree: hybrid memory monitoring for C. In *Proceedings of the Annual ACM Symposium on Applied Computing*, pages 1765–1772. ACM, April 2015.

[18] A. Jakobsson, N. Kosmatov, and J. Signoles. Fast as a shadow, expressive as a tree: Optimized memory monitoring for C. *Science of Computer Programming*, 132, Part 2:226 – 246, 2016. Special Issue on Software Verification and Testing (SAC-SVT'15).

[19] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX, June 2002.

[20] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26. Linköping University Electronic Press, September 1997.

[21] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[22] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *Proceedings of the International Conference on Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 167–182. Springer, September 2013.

[23] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 147–163. USENIX Association, October 2014.

[24] A. Kwon, U. Dhawan, J. M. Smith, T. F. K. 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 *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 721–732. ACM, November 2013.

[25] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258. ACM, June 2009.

[26] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[27] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Notices*, 42(6):89–100, June 2007.

[29] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, December 2006.

[30] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, December 2004.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–319. USENIX Association, June 2012.

[32] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for threadsanitizer. In *Proceedings of the International Conference on Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, September 2011.

[33] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX Annual Technical Conference*, pages 17–30. USENIX, 2005.

[34] SGCheck: An experimental stack and global array overrun detector. `http://valgrind.org/docs/manual/sg-manual.html`.

[35] Standard Performance Evaluation Corporation. SPEC CPU, 2006. `http://www.spec.org/benchmarks.html`.

[36] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE Computer Society, February 2015.

[37] W. Szpankowski. Patricia tries again revisited. *Journal of the ACM*, 37(4):691–711, October 1990.

[38] Projects using Valgrind. `http://valgrind.org/gallery/users.html`.

[39] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, September 2012.

[40] K. Vorobyov, P. Krishnan, and P. Stocks. A dynamic approach to locating memory leaks. In *Proceedings of the IFIP International Conference on Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 255–270. Springer, November 2013.

[41] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126. ACM, October - November 2004.

[42] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 116–127. ACM, October 2007.