Formal Methods in Computer-Aided Design 2022

Bounded Model Checking for LLVM

Siddharth Priya
University of Waterloo

Yusen Su
University of Waterloo

Yuyan Bao
University of Waterloo

Xiang Zhou
University of Waterloo

Yakir Vizel
The Technion

Arie Gurfinkel
University of Waterloo

***Abstract*—Bounded Model Checking (BMC) is an effective and precise static analysis technique that reduces program verification to satisfiability (SAT) solving. In this paper, we present the design and implementation of a new BMC engine (SEABMC) in the SEAHORN verification framework for LLVM. SEABMC precisely models arithmetic, pointer, and memory operations of LLVM. Our key design innovation is to structure verification condition generation around a series of transformations, starting with a custom IR (called SEA-IR) that explicitly purifies all memory operations by explicating dependencies between them. This transformation-based approach enables supporting many different styles of verification conditions. To support memory safety checking, we extend our base approach with fat pointers and shadow bits of memory to keep track of metadata, such as the size of a pointed-to object. To evaluate SEABMC, we have used it to verify `aws-c-common` library from AWS. We report on the effect of different encoding options with different SMT solvers, and also compare with CBMC, SMACK, KLEE and SYMBIOTIC. We show that SEABMC is capable of providing order of magnitude improvement compared with state-of-the-art.**

I. INTRODUCTION

Bounded Model Checking (BMC) is an effective technique for precise software static analysis. It encodes a bounded (i.e., loop- and recursion-free) program P with assertions into a verification condition VC in (propositional) logic, such that VC is satisfiable iff P has an execution that violates an assertion. The satisfiability of VC is decided by a SAT-solver (or, more commonly, by an SMT-solver). BMC can be extremely precise, including path-sensitivity, bit-precision, and precise memory model. Its key weakness is scalability – precise reasoning requires careful selection of what details to include into the analysis.

A BMC engine can be implemented directly at the level of program source code, as best illustrated by CBMC [1] – the oldest and most mature BMC for C. This allows verifying absence of undefined behaviour and other source-level properties, and improves error reporting since it can be done at the source level. However, this complicates the implementation because modern programming languages are incredibly complex. Moreover, most industrial code relies on *de-facto*, rather than the standard language semantics [2] and on non-standard features that are supported by mainstream compilers. An alternative is to implement BMC on an intermediate representation (IR) of a compiler. LLVM IR [3], called *bitcode*, is a common choice. This simplifies implementation to focus only on capturing semantics of the IR, allows sharing infrastructure with the compiler, simplifies integration of verification into current build systems, and simplifies supporting multiple source languages (e.g., SMACK [4] supports 8 languages [5]). This is the approach we take in this paper.

Over the years, there have been multiple BMC tools developed for LLVM, including SEAHORN (that we build on), SMACK, and LLBMC [6]. However, the issue still remains that existing tools are either not maintained, commercial (and not publicly available, e.g. LLBMC), or are not effective at bit- and memory-precise reasoning (SEAHORN and SMACK). Our goal is to address this deficiency, while re-examining and re-evaluating many of the design decisions. Thus, while BMC is a mature technique, we have two objectives. First, we want different strategies for generating verification conditions (VCGen) through program transformations. This allows us to examine which encoding works best in practice for production code, and why. Second, we want to provide mechanisms to express safety properties, e.g. memory safety, succinctly. In accomplishing these objectives, we believe that we have identified a new interesting point in the design space.

For our first objective, we propose a new pipeline. A source program is translated to a new IR, called SEA-IR, that extends LLVM IR, with explicit dependency between memory operations. This, effectively, purifies memory operations, i.e., there is no global memory, and no side-effects. A SEA-IR program then goes through a series of program transformations for VCGen. The program is progressively reduced to a pure data-flow form in which all instructions execute in parallel, and is only then, converted to SMT-LIB supported logic. This allows experimenting with different strategies of VCGen by controlling these transformations. For example, we can generate VCs using a control flow representation of the program like DAFNY [7] or a pure data flow representation like CBMC. VCs depend on memory representation. Thus, we explore two different forms of representing memory content: lambda-based [8] that represents memory as nested ITE-expressions[1], and array-based that uses SMT theory of arrays [9]. In particular, lambda-based representation allows precise and efficient modelling of wide memory operations such as `memcpy`. We

[1]ITE stands for If-Then-Else.

https://doi.org/

also explore the space of memory models between flat memory in which memory is a flat array, and an object memory where memory is represented by a set of arrays.

To improve checking for safety properties, our second objective, we attach additional information to pointers (so called *fat*) and to memory (so called *shadow*). This simplifies tracking of various program metadata for modelling safety properties. As an example, we can use fat pointers to check for out of bounds array access and shadow memory to check for immutability of read only memory. While existing tools report memory safety analysis, SEABMC can capture metadata of arbitrary size since we are not constrained by concrete pointer or memory width. Additionally, we model pointer provenance. This allows us to catch out-of-bounds accesses which might be missed by tools like LLBMC and ASAN [10].

We evaluate SEABMC on verification tasks of `aws-c-common` C library developed by Amazon Web Services (AWS). The library is a collection of common data-structures for C (including buffers, arrays, lists, etc.). We chose it for several reasons. First of all, it has been recently verified using CBMC. Thus, it includes many meaningful verification tasks. Second, it is a live industrial project, thus, it provides an example of how to integrate SEABMC into a real project, and shows that SEABMC supports all of the necessary language features. Third, it provides an opportunity to compare head-to-head against a mature tool (CBMC) on industrial code. We feel this is a more interesting comparison than, for example, comparing on isolated verification benchmarks of SVCOMP [11]. We show that SEABMC is an order of magnitude faster than CBMC, and outperforms three mature LLVM-based tools: SMACK, SYMBIOTIC [12] and KLEE [13]. Note that we focus on SEABMC design and performance. An extensive case study comparing different *kinds* of verification tools on `aws-c-common` is available in [14].

In summary, this paper makes the following contributions: an IR, SEA-IR, for LLVM bitcode that purifies memory operations; a VCGen that combines program transformations with encoding into logic allowing for many different styles of VCs; a memory model that combines fat-pointers with shadow-memory to represent metadata; an open-sourced BMC tool; and, a thorough evaluation against the state-of-the-art verification tools on production C code.

II. GENERATING VERIFICATION CONDITIONS

This section presents our main verification condition generation (VCGen) algorithm. We start with a new intermediate representation, that we call SEA-IR. This representation extends LLVM bitcode with purified memory operations. We then describe a series of transformations that transform a program in SEA-IR to a pure data-flow (PD) form where no part of computation depends on control. Each transformation progressively simplifies the program for generating verification conditions. The PD form is one from which verification conditions can be generated in the most straightforward way. Finally, we show how PD programs can be converted to verification conditions in SMT-LIB. In this section, we assume that the input program contains only one function, no loops or global variables. In practice, this is achieved by inlining all functions, unrolling loops to a fixed depth, and eliminating global variables. The loop unroll bound is often detected automatically, but can also be set by the user.

SEA-IR SEABMC transforms LLVM bitcode to an intermediate representation, called SEA-IR, that extends LLVM bitcode by making dependency information between memory operations explicit. In LLVM IR, this information does not exist in the program. Fig. 1 shows the simplified syntax of SEA-IR. Here, we present a simplified version with many features removed, e.g., types, expressions, function calls, etc. However, we assume that the type of each register is known (but not shown). We use `R` to represent a scalar register, `P` for a pointer register and `M` for a memory register. A legal SEA-IR program is assumed to be in a Static Single Assignment (SSA) form with all registers are assigned before use, all expressions well-typed and a program always ending with a **`halt`**.

We use the term *object* to refer to an allocated sequence of bytes in memory. Interestingly, we do not use a single addressable memory that maps from addresses to values. Instead, a SEA-IR program uses a set of memory regions or *memories*, which collectively contains all objects in a program. Each memory, in-turn, contains a subset of objects used in the program. To maintain compatibility with de-facto semantics, addresses are assigned from a single address space and are, thus, globally unique. To aid program analysis, all memories are pure: storing in memory creates a new memory i.e., *definition*; loading from a memory is a *use*. This def-use scheme [15] is known as MemorySSA in LLVM. Partitioning memory into multiple memories relieves the SMT-solver from some of the alias analysis reasoning.

To explain SEA-IR, we use a simple C program in Fig. 2. The program initializes variable `x` with a non-deterministic 8-bit integer obtained by the return value of function `nd_char()`. The value of x is further constrained by the **`assume`**, such that `x > 0 && x < 10`. Then, the program non-deterministically allocates 1- or 2-byte memory region and assigns the address to the variable `p`. The first byte that p points to is assigned by the value of x. The second byte (if any) is assigned 0. For the moment, ignore that the second assignment might be undefined behaviour (we expand on this in Sec III). Finally, the two **`assert`**s describe the post-condition.

Fig. 3a shows the SEA-IR program transformed from the C program. In this presentation, we do not strictly follow the syntax of SEA-IR. For example, we allow immediate values to appear in place of registers, and write expressions in infix form. The program is a single function **`main`**, which consists of four basic blocks labeled by `BB0`, `BB1`, `BB2` and `BB3`. A basic block consists of a label, zero or more `PHI`-statements, one or more statements, an optional branch statement or a **`halt`**.

A SEA-IR program has two types of registers: scalar registers and memory registers. Scalar registers store values of basic datatype – integers and pointers. Memory registers store memory regions, and map from addresses to values. Each

PR	::=	`fun main(){`BB$^+$`}`
BB	::=	L : PHI* S$^+$ (BR | `halt`)
BR	::=	`br` E, L, L | `br` L
PHI	::=	R = `phi [`R, L`](,[`R, L`])`* | M = `phi [`M, L`](,[`M, L`])`* | P = `phi [`P, L`](,[`P, L`])`*
S	::=	RDEF | MDEF | VS
RDEF	::=	R = E | P, M = `alloca` R, M | P, M = `malloc` R, M | R = `load` P, M | P = `load` P, M | M = `free` P, M
MDEF	::=	M = `store` R, P, M | M = `store` P, P, M
VS	::=	`assert` R | `assume` R

Fig. 1: Simplified grammar of SEA-IR, where `E`, `L` `R` and `M` are expressions, labels, scalar (or pointer) registers and memory registers, respectively.

```
1  int main() {
2    uint8_t x = nd_char();
3    assume(x > 0 && x < 10);
4    uint8_t *p = nd_bool() ? malloc(2*sizeof(uint8_t))
5                           : malloc(sizeof(uint8_t));
6    *p = x;
7    *(p + 1) = 0;
8    assert(0 < *p && *p < 10);
9    assert(*(p + 1) == 0); // potential UB
10   return 0;
11 }
```

Fig. 2: An example C program.

memory register maps to a unique *memory* and we use memory register and memory interchangeably. For example, in Fig. 3a, `R0` is a scalar register which stores an integer and `P1` is a scalar register for a pointer. `M0` and `M1` are memory registers. Since each program is finite, the number of registers is finite as well.

An assignment statement defines the register by the value of a given expression. We assume that expressions include the usual set of operations, e.g., arithmetic, bitwise operations, cast operations and pointer arithmetic. For example, in `BB0` of Fig. 3a, `R2 = R0 < 10` defines the value of register `R2` by the value of the expression `R0 < 10`, where $<$ is an unsigned 8-bit less-than operator.

A **`phi`** selects a value from a list of values when a control flow merges. For example, `M3 = phi[M1,BB1],[M2,BB2]` in `BB3` of Fig. 3a assigns `M1` (`M2`) to `M3` if the previously executed basic block was `BB1` (`BB2`).

SEA-IR provides **`alloca`** and **`malloc`** instructions to allocate memory on the stack and the heap, respectively. A given number of bytes are allocated in memory on RHS of the statement, defining a new memory on the LHS. While the allocation does not change memory, it does define it. This is explained in Sec. III. Consider `P1, M1 = malloc 2, M0` in `BB1` of Fig. 3a. It allocates 2 bytes (on the heap) in memory `M0`, defines memory `M1` and a fresh pointer in `P1`.

A **`store`**, e.g., `M5 = store 0, P5, M4` in `BB3`, defines memory `M5` by writing the value 0 to the address pointed-to by the pointer register `P5` in memory `M4`. Note that the instruction is pure; i.e., all effects of the instructions are on the output registers only. The result of the modification is in `M5`, while `M4` is unchanged. Similarly, a **`load`** reads the value pointed-to by a pointer register in memory register `M`, and assigns the value to a new register. **`assert`** and **`assume`** are the usual verification statements for assertions and assumptions, respectively.

Program Transformation Before generating verification conditions, a series of program transformations, as given below, are applied to a SEA-IR program.

Single Assert Form. A program is in a Single Assert (SA) form if it only contains one **`assert`**, which appears as the last instruction (before **`halt`**) in the last block of a program. Fig. 3b shows the code in a SA form transformed from the one in Fig. 3a, where an `ERR` label is added to the original code, and denotes an error state. In `BB3`, **`assert`** `R6` is transformed into **`br`** `R6, BB4, ERR`, meaning that if `R6` is false, then the program's execution trace is diverted to `ERR`. Similarly, **`assert`** `0 = 0` in `BB3` is transformed into **`assume`** `0 != 0` and **`br`** `ERR`.

Single Assume Single Assert (SASA) Form. A program is in SASA form if it is in SA form, and contains a single **`assume`** immediately followed by a single **`assert`**. For example, the two definition of registers `R1` and `R2` in `BB0` of Fig. 3b are combined into one definition of `R1` in Fig. 3c, where the two boolean expressions are combined by a conjunction. A **`phi`**-statement, `A = phi [R6,BB4],[R1,BB3]`, is added to `ERR`, so that register `A` tracks the value of the conjunction. The **`assume`** ensures that `A` is true prior to the assertion.

Gated Single Static Assignment Form. A program in SASA form is further transformed into a Gated Single Static Assignment (GSSA) form, where **`phi`**-functions are replaced by **`select`** expressions[2]. For example, **`phi`** `[M1,BB1], [M2,BB2]` in `ERR` of Fig. 3c is transformed into **`select`** `R2, M1, M2` in Fig. 3d, where `R2` is the condition that the program trace is diverted to `BB1` or `BB2`.

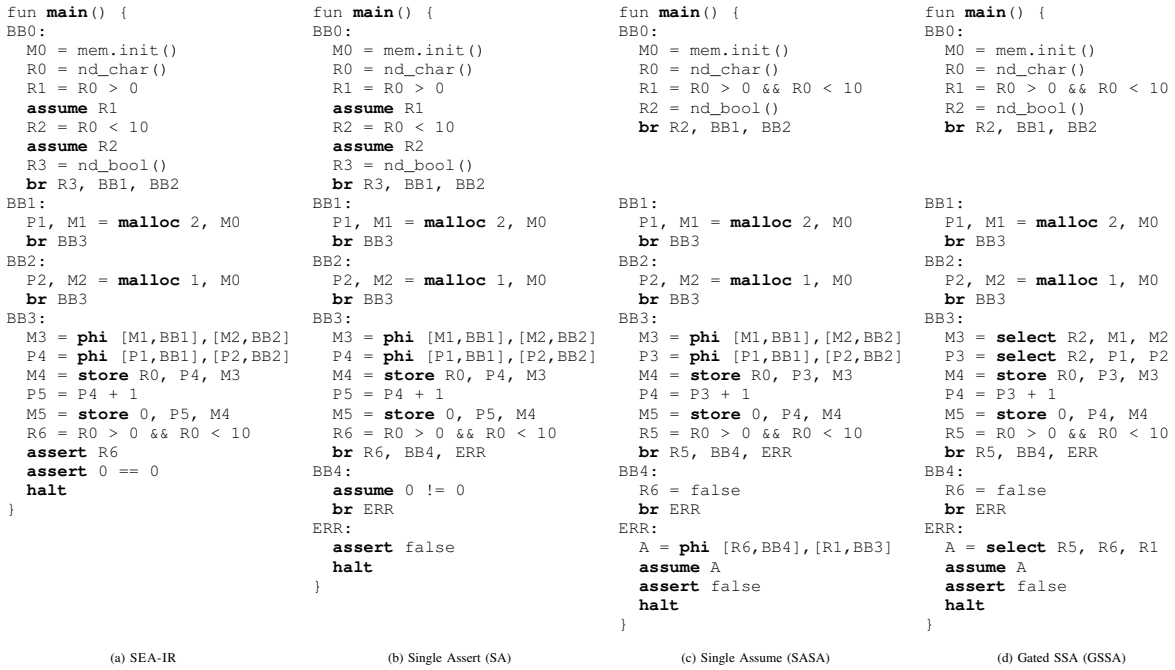
Pure Dataflow Form. A (loop-free) program is in a Pure Dataflow (PD) form if it is in GSSA form and contains a single basic block. As shown in Fig. 4a, all the labels and **`br`** are removed from Fig. 3d, and the five basic blocks are merged into one single basic block.

Reduced Pure Dataflow Form. A program is in a reduced PD form if every definition appears on a def-use chain of either **`assume`** or **`assert`**. Each such definition is said to be in the cone of influence (COI). In Fig. 4a, the highlighted code is not in the cone of influence and is not considered.

A reduced PD program has no control dependencies. It is essentially a sequence of equations with two side-conditions determined by **`assume`** and **`assert`**. All definitions are used, directly, or indirectly, by either **`assume`** or **`assert`** (or both). Now, generating VC implies mapping each definition into a logic equation.

Verification Condition Generation We now describe the translation function sym that encodes a program into a VC. Throughout the section, we illustrate sym using the program in Fig. 4a and the corresponding VC in Fig. 4b.

[2]In LLVM, **`select`** is the usual ternary ITE such as `a ? c : b` in C.



```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  P4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, P4, M3
  P5 = P4 + 1
  M5 = store 0, P5, M4
  R6 = R0 > 0 && R0 < 10
  assert R6
  assert 0 == 0
  halt
}
```

(a) SEA-IR

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0
  assume R1
  R2 = R0 < 10
  assume R2
  R3 = nd_bool()
  br R3, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  P4 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, P4, M3
  P5 = P4 + 1
  M5 = store 0, P5, M4
  R6 = R0 > 0 && R0 < 10
  br R6, BB4, ERR
BB4:
  assume 0 != 0
  br ERR
ERR:
  assert false
  halt
}
```

(b) Single Assert (SA)

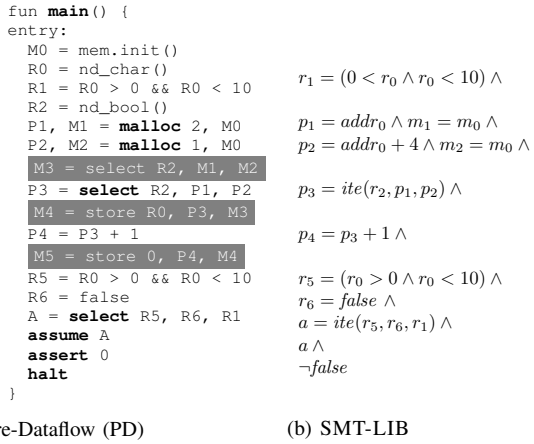
```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = phi [M1,BB1],[M2,BB2]
  P3 = phi [P1,BB1],[P2,BB2]
  M4 = store R0, P3, M3
  P4 = P3 + 1
  M5 = store 0, P4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = phi [R6,BB4],[R1,BB3]
  assume A
  assert false
  halt
}
```

(c) Single Assume (SASA)

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  br R2, BB1, BB2
BB1:
  P1, M1 = malloc 2, M0
  br BB3
BB2:
  P2, M2 = malloc 1, M0
  br BB3
BB3:
  M3 = select R2, M1, M2
  P3 = select R2, P1, P2
  M4 = store R0, P3, M3
  P4 = P3 + 1
  M5 = store 0, P4, M4
  R5 = R0 > 0 && R0 < 10
  br R5, BB4, ERR
BB4:
  R6 = false
  br ERR
ERR:
  A = select R5, R6, R1
  assume A
  assert false
  halt
}
```

(d) Gated SSA (GSSA)

Fig. 3: Program from Fig. 2 in: (a) SEA-IR, (b) SA, (c) SASA, and (d) GSSA forms.



```
fun main() {
entry:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  P1, M1 = malloc 2, M0
  P2, M2 = malloc 1, M0
  M3 = select R2, M1, M2
  P3 = select R2, P1, P2
  M4 = store R0, P3, M3
  P4 = P3 + 1
  M5 = store 0, P4, M4
  R5 = R0 > 0 && R0 < 10
  R6 = false
  A = select R5, R6, R1
  assume A
  assert 0
  halt
}
```

(a) Pure-Dataflow (PD)

$$
\begin{aligned}
& r_1 = (0 < r_0 \wedge r_0 < 10) \wedge \\
& p_1 = addr_0 \wedge m_1 = m_0 \wedge \\
& p_2 = addr_0 + 4 \wedge m_2 = m_0 \wedge \\
& p_3 = ite(r_2, p_1, p_2) \wedge \\
& p_4 = p_3 + 1 \wedge \\
& r_5 = (r_0 > 0 \wedge r_0 < 10) \wedge \\
& r_6 = false \wedge \\
& a = ite(r_5, r_6, r_1) \wedge \\
& a \wedge \\
& \neg false
\end{aligned}
$$

(b) SMT-LIB

Fig. 4: Program from Fig. 2 in PD and SMT-LIB forms. The highlighted lines are removed from the program.

The input to sym is a SEA-IR program in a reduced PD form, and the output is a SMT-LIB program. For simplicity of presentation, we assume that two fundamental sorts are used in the encoding: bit-vector of 64 bits, $bv(64)$, and a map between bit-vectors, $bv(64) \rightarrow bv(64)$.[3] In addition, we use the following helper sorts: $scalr : bv(64)$, $ptrs : scalr$, and $mems : bv(64) \rightarrow bv(64)$, where $scalr$ is sorts of scalars, $ptrs$ of pointers, and $mems$ of memories.

sym is defined recursively, bottom up, on the abstract syntax tree of SEA-IR. First, each register, `R`, is mapped to a symbolic constant $sym(\texttt{R})$ of an appropriate sort. To simplify the presentation, we use a lower-case math font for constants corresponding to the register. For example, in Fig. 4a, $sym(\texttt{R0})$ is r_0 of $scalr$ sort, $sym(\texttt{P2})$ is p_2 of $ptrs$ sort, and $sym(\texttt{M0})$ is m_0 of $mems$ sort, respectively.

[3]In practice, SEABMC supports multiple bit-widths for scalars, and different ranges for values for maps.

$$
\begin{aligned}
& sym(\texttt{R = E}) \triangleq r = e \quad sym(\textbf{assume } \texttt{R}) \triangleq r \quad sym(\textbf{assert } \texttt{R}) \triangleq \neg r \\
& sym(\texttt{M1 = } \textbf{store } \texttt{R1,P2,M0}) \triangleq m_1 = write(m_0, r_1, p_2) \\
& \qquad sym(\texttt{R1 = } \textbf{load } \texttt{P0,M}) \triangleq p_1 = read(m, p_0) \\
& sym(\texttt{P1,M1 = } \textbf{alloca } \texttt{R0,M0}) \triangleq p_1 = alloc(\textbf{alloca } \texttt{R0,M0}) \ \wedge \ m_1 = m_0 \\
& sym(\texttt{P1,M1 = } \textbf{malloc } \texttt{R0,M0}) \triangleq p_1 = alloc(\textbf{malloc } \texttt{R0,M0}) \ \wedge \ m_1 = m_0
\end{aligned}
$$

Fig. 5: Definition of sym.

Second, each expression `E` in SEA-IR is mapped into a corresponding SMT-LIB expression $sym(\texttt{E})$. We omit the details of this step since they are fairly standard. For example, a **`select`** is translated into an ite, scalar addition, such as `R9 + 1` is translated into bit-vector addition `bvadd`, etc. Pointer manipulating expressions, such as pointer arithmetic (**`gep`**) and pointer-to-integer cast (**`ptoi`**) are described in Sec. III.

Finally, sym translates each statement into an equality. For example, `R = E` is translated into $r = e$, where e is $sym(\texttt{E})$. For example, in Fig. 4a, `A = select R5,R6,R1` is translated into $a = ite(r_5, r_6, r_1)$ in Fig. 4b.

Translating **`alloca`** and **`malloc`** requires a memory allocator. We parameterize sym by an allocation function $alloc : A \rightarrow ptrs$ that maps allocation expressions in A to values of pointer sort. For example, in Fig. 5, `P1, M1 = alloca R0, M0` is translated into $p_1 = alloc(\textbf{alloca } \texttt{R0 M0}) \ \wedge \ m_1 = m_0$, and is reduced to $p_1 = addr_0 \wedge m_1 = m_0$, where $addr_0$ is the return value of $alloc$.

$$\forall a \in A \cdot size(a) \text{ is known} \quad \forall a \in A \cdot (alloc(a) \bmod align(a)) = 0$$
$$\forall a_1 \neq a_2 \in A \cdot (alloc(a_1) + size(a_1) \leq alloc(a_2)) \vee (alloc(a_2) + size(a_2) \leq alloc(a_1))$$

Fig. 6: Specifications for *size*, *align*, and *alloc*.

	Array	λ
$read(m, p_0)$	**select** m p_0	$m(p_0)$
$write(m_0, r_1, p_2)$	**store** m_0 r_1 p_2	$\lambda x.ite(x = p_2, r_1, m_0(x))$

Fig. 7: Translation of *read* and *write*.

For sym, $alloc$ must satisfy the basic specifications of a memory allocator. The spec is formalized in Fig. 6, where $size$ and $align$ return the size and alignment of each allocation expression in A. Intuitively, each allocated segment must have a statically known bound on size, all pointers returned by an allocation are aligned, and all allocations are mutually disjoint. For example, in Fig. 4a, the memory allocations in `P2, M1 = malloc 2, M0` and `P1, M2 = malloc 1, M0` are guaranteed to be disjoint since Fig. 4b adds a constraint that $p_1 = addr_0 \wedge p_2 = addr_0 + 4$. In practice, we also enforce that stack allocations (**alloca**) return high addresses, and heap allocations (**malloc**) return low addresses. Other constraints, such as separating kernel- and user-space addresses can be easily added.

The semantics for memory operations depends on the representation of memories (see Sec. III). We use two functions, $read$ and $write$, to encapsulate the actual translation when defining the meaning of **load** and **store**, respectively. The function $read(m, p)$ represents the value of the memory register m at index p. The function $write(m, r_1, p_2)$ represents a new memory obtained by writing the value r_1 at index p_2 in m. In Fig. 5, **load** `P0, M` and **store** `R1, P2, M0` are translated into $read(m, p_0)$, and $write(m_0, r_1, p_2)$, respectively.

SEABMC has two memory representations: *Arrays* and *Lambdas*.

Arrays. Memories are modeled by an SMT-LIB theory of extensional arrays ArraysEx[4]. A memory register `M` is mapped to a symbolic constant m, where m is of sort $mems$. As shown in Fig. 7, a *write* is translated into an ArrayEx store, and a *read* is translated into an ArrayEx select.

Lambdas. Memories are modelled by λ-functions of the form $\lambda x.e$, where e is an expression with free occurrences of x. A memory register `M` is translated into an uninterpreted function m of sort $mems$. As shown in Fig. 7, $read(m, r_0)$ is translated into a function application $m(r_0)$, and $write(m_0, r_1, p_2)$ is translated into a new λ-function, $\lambda x.ite(x = p_2, r_1, m_0)$. In the final VC, function applications are β-reduced to substitute formal arguments with actual parameters. Thus, the VC only has ites, and does not require ArrayEx support in the SMT-solver.

[4]http://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

RDEF ::= R = **isderef** R, R | R = **isalloc** R, M | R = **ismod** R, M

Fig. 8: SEA-IR syntax for memory safety.

```
fun main() {
BB0:
  M0 = mem.init()
  R0 = nd_char()
  R1 = R0 > 0 && R0 < 10
  R2 = nd_bool()
  P1, M1 = malloc 2, M0
  P2, M2 = malloc 1, M0
  M3 = select R2, M1, M2
  P3 = select R2, P1, P2
  R4 = isderef R3, 1
  M4 = store R0, P3, M3
  P5 = gep P3, 1
  R6 = isderef P5, 1
  M5 = store 0, P5, M4
  R7 = R0 > 0 && R0 < 10
  R8 = false
  A0 = select R7, R8, R1
  A1 = select R6, A0, R1
  A2 = select R4, A1, R1
  assume(A2)
  assert(0)
  halt
}
```

(a) Pure-Dataflow (PD)

$r_1 = r_0 > 0 \wedge r_0 < 10 \wedge$
$p_1.base = addr_0 \wedge p_1.offset = 0 \wedge p_1.size = 4 \wedge m_1 = m_0 \wedge$
$p_2.base = addr_0 + 4 \wedge p_2.offset = 0 \wedge p_2.size = 4 \wedge m_2 = m_0 \wedge$
$p_3 = ite(r_5, p_1, p_2) \wedge$
$r_4 = 0 \leq (1 + p_3.offset) < p_3.size \wedge$
$p_5.base = r_3.base \wedge p_5.offset = r_3.offset + 1 \wedge p_5.size = r_3.size \wedge$
$r_6 = 0 \leq (1 + p_5.offset) < p_5.size \wedge$
$r_7 = r_0 > 0 \wedge r_0 < 10 \wedge$
$r_8 = false \wedge$
$a_0 = ite(r_7, r_8, r_1) \wedge$
$a_1 = ite(r_6, a_0, r_1) \wedge$
$a_2 = ite(r_4, a_1, r_1) \wedge$
$a_2 \wedge$
$\neg false$

(b) VC in SMT-LIB

Fig. 9: Program from Fig. 2 in PD and SMT-LIB forms. The **isderef** instruction checks for spatial memory safety.

Overall, for a program `P` in a reduced PD form with a sequence of statements $\mathtt{S}_0 \cdots \mathtt{S}_k$, followed by **assume** `R0` and **assert** `R1`, $sym(\mathtt{P})$ is defined as follows:

$$sym(\mathtt{P}) \triangleq \left(\bigwedge_{0 \leq i \leq k} sym(\mathtt{S}_i) \right) \wedge sym(\mathtt{R0}) \wedge sym(\mathtt{R1}).$$

For example, the VC for a program in Fig. 4a is shown in Fig. 4b. Definitions in Fig. 4a are translated into a conjunction of equalities, and **assert** `0` is translated into $\neg false$. The VC is *unsatisfiable* since `A` evaluates to $false$.

Theorem 1: $sym(\mathtt{P})$ is satisfiable iff `P` has an execution that satisfies the assumption and violates the assertion.

III. VERIFYING MEMORY SAFETY

In most languages, including C, memory safety is difficult to specify directly. To make such specifications possible, we use fat pointers [16] and shadow memory to keep metadata about pointers and memory, respectively. Moreover, we present a general extension of both memory and pointer semantics.

Intuitively, we want to represent each fat pointer as a tuple of values that collectively represent the value of the pointer and all the metadata (i.e., *fat*) that is cached at it. We do not put restrictions on the number of values nor their sorts. However, we assume that there is a function $addr$ that maps a pointer to an expression representing an address. Thus, for a pointer register `P`, $sym(\mathtt{P})$ is a tuple $\langle t_1, \ldots, t_j \rangle$ of j constants that represents the pointer, and $addr(\langle t_1, \ldots, t_j \rangle)$ is an address of that pointer. For example, a common case is to use the first element of the tuple to represent the address: $addr(\langle t_1, \ldots, t_j \rangle) = t_1$. Fig. 13 presents a small program (on the left) that writes a fat pointer `P0` to memory at address `P1`. Memory is divided into five parts with val memory used

$sym(\texttt{P1,M1 = malloc R0,M0)}) \triangleq$
$\quad p_1 = alloc(\texttt{malloc R0, M0}) \wedge m_1 = alloc_{sh}(m_0, p_1)$
$sym(\texttt{M1 = free P0,M0}) \triangleq m_1 = free_{sh}(m_0, p_0)$
$sym(\texttt{MR = store R1,P2,M1)}) \triangleq$
$\quad \langle m_{r1}, \ldots, m_{rj} \rangle = \langle write(m_0.1, r_1, addr(p_2.1)), \ldots,$
$\qquad\qquad write(m_0.j, r_1, addr(p_2.j)) \rangle \wedge$
$\quad \langle m1_{j+1}, \ldots, m1_k \rangle = store_{sh}(\langle m_{0j+1}, \ldots, m_{0k} \rangle, p_2)$
$sym(\texttt{R1 = load P0,M0}) \triangleq$
$\quad r_1 = \langle read(m_0.1, addr(p_0)), \ldots, read(m_0.j, addr(p_0)) \rangle$

Fig. 10: Memory-safety aware VCGen semantics.

$alloc_{sh}(m, p) \triangleq$
$\quad \langle m.val, m.offset, m.size, write(m.alloc, 1, p.base), m.mod \rangle$
$free_{sh}(m, p) \triangleq$
$\quad \langle m.val, m.offset, m.size, write(m.alloc, 0, p.base), m.mod \rangle$
$store_{sh}(\langle m.alloc, m.mod \rangle, p) \triangleq$
$\quad \langle m.alloc, write(m.mod, 1, p.base) \rangle$

Fig. 11: Shadow memory semantics for memory safety.

to store the actual program data. Here, *val* stores the *base* value of the fat pointer and *offset* and *size* store the fat. Memory operations are tracked by *alloc* and *mod* memory that mark whether an address is allocated and whether it has been written to, respectively. Fig. 13 shows the memory state after the **store** operation. Both *alloc* and *mod* are set to 1 because P1 is allocated and has been modified.

Formally, we re-define *ptrs* to be a tuple of sorts, written as $\langle s_1, \ldots, s_j \rangle$. We say that a tuple $\tau = \langle c_1, \ldots, c_p \rangle$ of p constants is of a tuple sort $\langle s_1, \ldots, s_p \rangle$ iff, for each $0 < i \leq p$, c_i is of sort s_i. Tuples of sorts, and tuples of constants are only present during VCGen, but not in the final verification condition. For that, we rewrite equality between two tuples as conjunction of equalities between their elements, and use $\tau.i$ for the ith element of tuple τ.

Similarly, we re-define *mems* for a memory register M to be a tuple of values that store the program and the shadow states. Thus, $sym(\texttt{M}) = \langle v_0, \ldots, v_k \rangle$, where each v_i is the sort $bv(64) \rightarrow bv(64)$. If a pointer is represented by a j-tuple, we assume that memory is represented by a k-tuple, with $k \geq j$, so that the first j entries in a memory register are wide enough to store the fat pointer. Specifically, we require that the sort of v_j is same as sort of t_j for $1 \leq j \leq k$.

We modify the semantics of **malloc** by storing meta data along with explicit program states. The modification is defined in Fig. 10 (M1 is now a memory tuple). The signature of *alloc* is unchanged, but now returns a fat pointer. Given a pointer p of sort *ptrs*, a function $size(\langle t_1, \ldots, t_j \rangle)$ returns the size of a memory object pointed-to by p. An additional function $alloc_{sh} : mems \rightarrow mems$ operates on shadow memory. The semantics of $alloc_{sh}$ and $free_{sh}$ is described later.

A **store** is divided into two parts. First is the store of the actual program data. Since the data can be of sort *scalr* or *ptrs*, a store of a k-tuple of data on memory m_0 is translated into k *writes*, on each element of $\langle m_{01}, \ldots, m_{0j} \rangle$.

$sym(\texttt{R1 = isderef P0 B}) \triangleq r_1 = 0 \leq p_0.offset < p_0.size$
$sym(\texttt{R1 = isalloc P0 M}) \triangleq r_1 = read(m.alloc, p_0.base)$
$sym(\texttt{R1 = ismod P0 M}) \triangleq r_1 = read(m.mod, p_0.base)$

Fig. 12: Semantics for verifiying memory safety.

```
fun main() {
BB0:
  M0 = mem.init()
  // addr=0x100
  P0, M1 = malloc 1, M0
  // addr=0x200
  P1, M2 = malloc 1, M1
  M3 = store P0, P1, M2
  halt
}
```

(a) a program

Memory

	val	offset	size	alloc	mod
0x100	0	1	1	1	

P1:

base	offset	size
0x200	0	1

(b) memory state

Fig. 13: Memory state M3 - when P0 is stored at location P1.

Second is updating metadata, done by $store_{sh}$ that works on $\langle m_{0j+1}, \ldots, m_{0k} \rangle$. The details of $store_{sh}$ are described later in this section. Similarly, a **load** expects to read $\langle m_{01}, \ldots, m_{0j} \rangle$ of sort *ptrs*. This allows representing arbitrary fat and shadows. We illustrate its specializations for memory safety next.

Spatial memory safety A program satisfies spatial memory safety iff every read and write is always inside an allocated object. A fat pointer is defined as a tuple of three constants $\langle s_1, s_2, s_3 \rangle$ denoted as $\langle base, offset, size \rangle$ for convenience. Here *base* is the start address of the object, *offset* is an index into the object, and *size* is its size. The address *addr* is given by $base + offset$.

With fat pointers, we introduce instructions for pointer arithmetic and pointer integer casts. The instruction **gep** is used for pointer arithmetic. Fig. 9a shows an example use in R5 = **gep** R3, 1. Here, semantically, a new pointer R5 is created that has the same *base* and *size* as R3, with *offset* incremented by 1. We also introduce **ptoi** instruction that casts a pointer to an integer by adding *offset* to *base*. For an integer to pointer cast, we use the **itop** instruction. This instruction sets *base* to the integer value and fat (i.e., metadata) to zero.

To assert that a pointer dereference is spatially safe, we provide an **isderef** instruction, whose semantics is shown in Fig. 12. For example, the program in Fig. 9a executes **assert**(0) as R6 = **isderef** R5, 1 evaluates to *false* causing A1 and A2 to evaluate to R1 and *true*, respectively. Thus, the VC in Fig. 9b is *satisfiable* which exposes the out of bounds error in Fig. 2 line 9. Note that this error is not caught by the VC in Sec. II. In SeaBmc, we automatically add **isderef** assertions before memory accesses. Many of such assertions are statically and, thus, cheaply resolved to *true* or *false* prior to SMT solving.

Note that SeaBmc semantics for spatial safety differs from LLBMC [17]. LLBMC treats only accesses to unallocated memory as unsafe. This implies that it is valid for a pointer to overflow into another object allocated just below or above. In SeaBmc, jumping across the allocated boundary is in-

valid. SEABMC also differs from CBMC in this regard. In CBMC [1], the pointer representation is fixed and a few bits in the pointer representation are reserved for fat data. These constraint the available address range. Additionally, only limited metadata can be stored in each pointer. In SEABMC, we support composite pointer representations that maintain parity with concrete pointer representation while allowing for rich metadata in the fat region of the pointer.

Temporal memory safety A program satisfies temporal memory safety iff it never does one of the following: **(UAF)** an object is used after it has been freed; and **(RO)** an object marked as read-only (by programmers) is modified. We detect a violation of memory safety by tracking the status of a memory object using shadow memory. Each memory is a tuple $\langle v_1, \ldots, v_5 \rangle$ of constants of sort $bv(64) \rightarrow bv(64)$, denoted $\langle val, offset, size, alloc, mod \rangle$, where $\langle val, offset, size \rangle$ maps to pointer data $\langle base, offset, size \rangle$, and $alloc$ and mod track the allocated and modified status of an object, respectively.

An object can be in allocated or freed state. To track allocated state, sym in Sec. II is extended for **`alloca`**, **`malloc`**, and **`free`**. The new semantics is shown in Fig. 10. The function $alloc_{sh} : mems \rightarrow mems$ is defined, for temporal memory safety, as shown in Fig. 11. Note that $alloc_{sh}(m, r)$ marks $m.mod$ memory only at the start of an object, i.e., $r.base$. For this reason it is necessary to use the fat pointer representation since it records the $base$ for every pointer. The **`isalloc`** instruction, shown in Fig. 8, is used to check the allocated state of an object at any point in the program. The semantics for **`isalloc`** is defined in Fig. 12.

A C program has no native mechanism for verifying that an object remains unmodified when passed to a function. To remedy this, we extend the semantics for **`store`** (see Fig. 10). The function $store_{sh} : mems \rightarrow mems$ is implemented for temporal memory safety (see Fig. 11). The **`ismod`** in Fig. 8 is used to check the read-only state of an object at any program point. The semantics for **`ismod`** is given in Fig. 12. We also provide a companion instruction `resetmod R, M` that resets $m.mod$ at address $r.base$ to zero. This allows initializing an object, resetting modified state, and then checking that the subsequent program does not modify the object. We track memory state only at object granularity, therefore, the current implementation is tied to using the fat pointer representation.

IV. EXPERIMENTS

In this section, we describe the evaluation of SEABMC[5] on verification tasks from `aws-c-common`. Each task verified post-conditions and memory safety of a single function from `aws-c-common`. Overall, there are 169 tasks in 20K LOC. Results and tasks are available at https://github.com/seahorn/verify-c-common[6]. We have chosen these tasks because they represent a real industrial use-case of BMC. We have adapted them from CBMC to be compatible with LLVM-based C verification tools. Note that here we focus on SEABMC performance. A detailed comparison of different *kinds* of verification tools on `aws-c-common` is presented in [14].

Comparing Different VCGen Strategies We evaluate the effectiveness of the different VCGen strategies by controlling which transformations are enabled. The main performance metric is *time solved* – the time to solve *all* solved tasks[7] (i.e., with timeout excluded). The time limit is 600s per task.

First, we evaluate the two memory representations: Arrays vs Lambdas. We use Z3 [18] and YICES2 [19] to account for the difference between SMT-solvers. The results are summarized in Tab. Ia. For Z3, we find that Arrays are less efficient than Lambdas. For YICES2, the results are comparable, suggesting that the choice of the representation is less important. Z3 with Lambdas is the overall winner, and we use it for the rest of the experiments.

Second, we evaluate the effectiveness of the transformations in Sec. II. The results are in Tab. Ib. Here, *optimal* means applying all of the transformation involved, plus eagerly simplifying VC during VCGen. β-reducing lambdas introduces many nested ITE-terms, so simplifying them early is useful.

To evaluate, we compare with 5 additional strategies by disabling some transformations: 1) rel_alloc – use $alloc$ that returns relative addresses from some symbolic start of stack and heap, rather than concrete addresses 2) flat_mem – one flat memory instead of using alias analysis to partition memory into disjoint memories as much as possible 3) no_coi – disable cone-of-influence 4) no_simp – disable eager simplification 5) p_cond – generate VC directly from SSA form by using path condition to encode **`phi`**-functions as in [6], [20]. Removing any of the transformations either noticeably degrades performance, or causes a timeout.

SEABMC supports memory word size of 1 byte ($bv(8)$), 4 bytes ($bv(32)$) and 8 bytes ($bv(64)$). The 1-byte words are most precise and support arbitrary memory accesses, while 8-byte words require aligned accesses. The comparison between the two is shown in Tab. Ic. Wider words significantly improve performance, but can be unsound for some benchmarks. By supporting both, SEABMC lets the user pick most appropriate choice per benchmarks. In other experiments, we adjust word size per individual benchmarks.[8]

Shadow memory performance A C program has no builtin mechanism for verifying that an object is not modified by a function. To overcome this limitation, the verification tasks in `aws-c-common` record the value of a byte from a non deterministic offset within an allocated object and then verify that this byte is unchanged in all executions. While this is a clever technique, setting it up in a verification task is complex. The **`ismod`** instruction added in SEABMC (see Sec. III) offers a user friendly alternative. We also found it to be more performant in the SEABMC implementation. We ported 70 tasks in `aws-c-common` to use **`ismod`**. Ported tasks ran 55% faster, on average, than their originals (see Tab. Id). This

[5]Source at https://github.com/seahorn/seahorn/tree/dev10.

[6]This website includes instructions for reproducing the experiments.

[7]This analysis uses 172 tasks instead of 169. 3 tasks are SEABMC specific.

[8]CBMC uses a similar per-benchmark configuration as well.

config	solver	unsat	timeout	failed	solved time(s)
array	z3	158	8	6	1 647
	yices2	170	0	2	1 016
lambdas	z3	172	0	0	**836**
	yices2	172	0	0	912

(a) Different memory representations.

config	unsat	timeout	solved time(s)	avg(s)	std(s)
optimal	172	0	836	5	10
rel_alloc	172	0	1 456	8	19
flat_mem	163	9	2 689	16	55
no_coi	170	2	849	5	10
no_simp	166	6	1 429	9	33
p_cond	170	2	659	4	6

(b) Different encodings.

word size	unsat	timeout	failed	solved time(s)
$bv(64)$	156	0	16	679
$bv(8)$	171	1	0	2 546

(c) Different word sizes.

config	unsat	solved time(s)
no shadow memory	70	143
shadow memory	70	**90**

(d) Different memory features.

TABLE I: Evaluations of different configuration.

strengthens the case for shadow memory from both usability and performance perspectives.

SEABMC vs. State-of-the-Art Overall, the results for our configurations in previous discussion suggest that the *optimal* strategy provides best performance in terms of precision and efficiency. We also consider four tools comparing against: CBMC [1], SMACK [4], KLEE [13], and SYMBIOTIC [12]. LLBMC is another interesting BMC tool, however, we decided to exclude it from comparisons due to the lack of an easily accessible public version[9] for user to reproduce LLBMC results. CBMC is, perhaps, the oldest and most well-known BMC for C programs (not based on LLVM). It is actively used by AWS, and was used for the verification of `aws-c-common`. SMACK is an LLVM-based BMC tool that uses Boogie [21] and Corral [4] for bounded and deductive verification. SYMBIOTIC is a KLEE-based tool that combines program instrumentation, slicing, and symbolic execution [22]. Both SMACK and SYMBIOTIC performed very well on the "SoftwareSystems" category in SV-COMP'21. KLEE is a LLVM-based symbolic execution tool that does not encode the VC in one shot but rather explores satisfiability of path conditions in a program one path-at-a-time. It is a practical alternative to BMC.

The results collected on an AMD Ryzen(TM) 5 5600X CPU with 32 GB memory are shown in Tab. II. Only SEABMC and CBMC solve all verification tasks from `aws-c-common`. SMACK in bit-precise mode times out on most instances, and in arithmetic mode times out on 20 and fails on 4. SYMBIOTIC times out on 5 and fails on 10. It is best-performing on `priority_queue` and `ring_buffer`. However, it also failed to detect seeded bugs[10], which questions its results. KLEE is particularly effective on `linked_list` – showing the benefit of exploring path-at-a-time, when number of paths is small.

Bugs found In [14], we discuss bugs found and reported to AWS. One example, in Fig. 14, concerns the `byte buffer` data structure that is defined as a length delimited byte string. Its data representation should be either the buffer (`buf`) is `NULL` or its capacity (`cap`) is 0 (not the `len` as defined in `BB_is_ok` Line 7). Under the correct model (a malloc that can potentially fail), SEABMC produces counter examples in 50 seconds, CBMC in 112 seconds. However, KLEE cannot

[9]LLBMC source code is not publicly available; Binary download on website is broken.

[10]Details at https://github.com/seahorn/verify-c-common/issues/124

```
1  typedef
2  struct byte_buf {
3    char* buf;
4    int len, cap;
5  } BB;
6  bool BB_is_ok(BB *b)
7  { return (b->len == 0
8          || b->buf);  }
```

Fig. 14: Incorrect `byte_buf` invariant

detect this bug since it needs an allocated buffer with an explicit size to proceed with analysis.

Overall, SEABMC outperforms competitors on most categories and in the overall run-time. Thus, we conclude that SEABMC is a highly efficient BMC engine.

We have compared SEABMC with tools from SV-COMP, but not with the benchmarks. There are two reasons. First, while a version of `aws-c-common` appears in SV-COMP, it is pre-processed with CBMC harnesses, and, therefore, includes undefined behaviors (e.g., uninitialized variables). This is not supported by SEABMC front-end. Second, we felt it is more important to validate tools in an actively developed code-base. Thus, we focused our effort on building an infrastructure for continuously verifying current `aws-c-common` using many existing tools, rather than integrating SEABMC into the rules of SV-COMP.

V. RELATED WORK

Bounded Software Model Checking is a mature program analysis technique. We briefly review only some of the closest related work. Over the years, there have been many model checking tools built on top of the LLVM platform. The closest to ours is the work of Babic [23] and LLBMC [17]. Similarly to [23], we rely on the Gated SSA form to remove all control dependence leaving only data-flows to be represented. However, our encoding is significantly simplified by an intermediate representation that purifies memory flows. Unfortunately, [23] has not been maintained making head-to-head comparison difficult.

We borrow the idea of using lambda-encoding for representing memory from LLBMC [17]. One important advantage of lambdas is that we can represent memory operations such as **`memcpy`** efficiently (while with arrays, these have to be unfolded). In particular, this allows for unbounded verification of loop-free programs that use these operations. The most significant difference from LLBMC is in our encoding of memory safety. In particular, we cache bounds information in the pointer, and check that every access is inside the

	Statistics		SeaBmc			CBMC			SMACK					Symbiotic					KLEE			
category	cnt	loc	avg (s)	std (s)	time (s)	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	fld/to	avg (s)	std (s)	time (s)	cnt	avg (s)	std (s)	time (s)
arithmetic	6	202	1	0	**3**	4	0	22	6	2/0	3	1	18	6	0/0	135	281	809	6	1	0	5
array	4	390	2	1	**7**	6	0	23	4	0/1	53	98	213	4	0/0	11	4	44	4	26	2	103
array_list	24	3,150	3	4	**71**	19	33	450	24	0/0	5	1	126	23	0/0	43	68	980	24	41	38	994
byte_buf	29	2,908	1	1	**29**	9	10	252	29	0/2	27	50	788	29	0/0	40	162	1,168	27	59	96	1,592
byte_cursor	24	2,365	1	0	**23**	6	3	153	16	0/2	32	66	519	17	0/0	7	4	125	17	10	11	169
hash_callback	3	347	6	5	18	8	5	25	3	0/0	4	2	**11**	3	0/0	40	62	120	3	50	38	151
hash_iter	4	708	9	15	**37**	10	6	39	4	0/0	91	58	363	3	0/1	37	44	112	3	14	6	41
hash_table	19	3,295	6	8	**105**	19	28	366	19	2/4	54	79	1,025	15	8/4	472	1,261	7,088	15	33	72	492
linked_list	18	2,127	2	2	37	33	112	595	18	0/5	96	91	1,735	18	0/0	8	5	143	18	1	0	**12**
others	2	31	0	0	**1**	4	0	7	1	0/0	2	0	2	1	0/0	5	0	5	1	1	0	**1**
priority_queue	15	3,004	14	22	202	286	700	4,284	15	0/1	20	50	307	15	0/0	10	20	**152**	15	32	8	473
ring_buffer	6	934	21	22	128	13	8	78	6	0/3	133	98	796	6	1/0	10	9	**63**	6	30	16	180
string	15	1,329	3	2	**49**	7	5	104	15	0/2	31	69	467	15	1/0	9	11	137	15	102	106	1,528
total	169	20,790			**710**			6,398		4/20			6,370		10/5			10,946				5,741

TABLE II: Verification results for SeaBmc, CBMC, SMACK, Symbiotic, and KLEE. Timeout for SMACK and SeaBmc is 200s, and 5,000s for Symbiotic. **cnt**, **fld**, **to**, **avg**, **std** and **time**, are the number of verification tasks, failed cases, timeout cases, average run-time, standard deviation, and total run-time in seconds, per category.

allocated memory object. In contrast, LLBMC assumes an arbitrary allocator and checks that all accesses are into some allocated memory, not necessarily into the expected object. Unfortunately, there is no public version of LLBMC available, precluding a head-to-head comparison.

SMACK [4], [5] is probably the most known BMC for LLVM. It is based on Boogie and Corral from Microsoft Research. It is most effective for arithmetic abstraction of software (i.e., abstracting machine integers by arbitrary precision integers). Its model for memory safety relies on complex encoding using universally quantified axioms in Boogie, leading to quantified reasoning in SMT. In contrast, our representation is tuned to perform well with modern SMT solvers. SMACK shares SeaDsa [24], [25] alias analysis with SeaBmc. DIVINE4 [26] is an explicit state model checker that also targets LLVM. However, it uses LLVM 7 which makes head-to-head comparison difficult. It targets parallel programs, which SeaBmc does not. For sequential programs, it is related to `libFuzzer` and KLEE that we compare with.

VI. Conclusion

We have presented the techniques behind SeaBmc, a new LLVM-base Bounded Model Checker for C. SeaBmc is path-sensitive, bit-precise, and provides a precise model of memory. It extends the traditional memory model with *fat* pointers and *shadow* memory that allow attaching metadata to pointers and memory. We have evaluated SeaBmc against CBMC, SMACK, Symbiotic, and KLEE and show significant performance improvements over the competition.

References

[1] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[2] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 1–15.

[3] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88.

[4] Z. Rakamaric and M. Emmi, "SMACK: decoupling source language details from verifier implementations," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 106–113.

[5] J. J. Garzella, M. S. Baranowski, S. He, and Z. Rakamaric, "Leveraging compiler intermediate representation for multi- and cross-language verification," in *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. Beyer and D. Zufferey, Eds., vol. 11990. Springer, 2020, pp. 90–111.

[6] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Joshi, P. Müller, and A. Podelski, Eds., vol. 7152. Springer, 2012, pp. 146–161.

[7] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. M. Clarke and A. Voronkov, Eds., vol. 6355. Springer, 2010, pp. 348–370. [Online]. Available: https://doi.org/10.1007/978-3-642-17511-4_20

[8] S. Falke, F. Merz, and C. Sinz, "Extending the theory of arrays: memset, memcpy, and beyond," in *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Springer, 2013, pp. 108–128. [Online]. Available: https://doi.org/10.1007/978-3-642-54108-7_6

[9] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[11] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 401–422.

[12] J. Slaby, J. Strejcek, and M. Trtík, "Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Piterman and S. A. Smolka, Eds., vol. 7795. Springer, 2013, pp. 630–632. [Online]. Available: https://doi.org/10.1007/978-3-642-36742-7_50

[13] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[14] S. Priya, X. Zhou, Y. Su, Y. Vizel, Y. Bao, and A. Gurfinkel, "Verifying verified code," in *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2021.

[15] F. C. Chow, S. Chan, S. Liu, R. Lo, and M. Streich, "Effective representation of aliases and indirect memory operations in SSA form," in *Compiler Construction, 6th International Conference, CC'96, Linköping, Sweden, April 24-26, 1996, Proceedings*, ser. Lecture Notes in Computer Science, T. Gyimóthy, Ed., vol. 1060. Springer, 1996, pp. 253–267.

[16] 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, June 10-15, 2002, Monterey, California, USA*, C. S. Ellis, Ed. USENIX, 2002, pp. 275–288.

[17] C. Sinz, S. Falke, and F. Merz, "A precise memory model for low-level bounded model checking," in *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*, R. Huuck, G. Klein, and B. Schlich, Eds. USENIX Association, 2010.

[18] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[19] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.

[20] A. Gurfinkel, S. Chaki, and S. Sapra, "Efficient predicate abstraction of program summaries," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 131–145.

[21] K. R. M. Leino, "This is Boogie 2," 2008.

[22] J. Slaby, J. Strejcek, and M. Trtík, "Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution," in *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, ser. Lecture Notes in Computer Science, M. Stoelinga and R. Pinger, Eds., vol. 7437. Springer, 2012, pp. 207–221. [Online]. Available: https://doi.org/10.1007/978-3-642-32469-7_14

[23] D. Babić, "Exploiting Structure for Scalable Software Verification," Ph.D. dissertation, University of British Columbia, Canada, 2008.

[24] A. Gurfinkel and J. A. Navas, "A context-sensitive memory model for verification of C/C++ programs," in *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, F. Ranzato, Ed., vol. 10422. Springer, 2017, pp. 148–168.

[25] J. Kuderski, J. A. Navas, and A. Gurfinkel, "Unification-based pointer analysis without oversharing," in *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, C. W. Barrett and J. Yang, Eds. IEEE, 2019, pp. 37–45.

[26] Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai, and V. Štill, "Model checking of C and C++ with DIVINE 4," in *Automated Technology for Verification and Analysis*, ser. LNCS, vol. 10482. Springer, 2017, pp. 201–207.

verification task	config	tool	run-time (s)
aws-array-list-erase	all	SEABMC	4
		CBMC	98
aws-array-list-erase	no-mem-safe	SEABMC	2
		CBMC	98
aws-array-list-erase	no-memmove	SEABMC	3
		CBMC	40

TABLE III: SEABMC vs. CBMC for `aws-array-list-erase`.

flag	meaning
`--unwind 1`	number of times to unwind loops
`--flush`	print to stdout
`--object-bits 8`	number of pointer bits to store meta information
`--malloc-may-fail`	malloc may fail
`--malloc-fail-null`	malloc may fail and return `NULL`
`--bounds_check`	check access is within bounds
`--pointer_check`	check access is within bounds

TABLE IV: CBMC options for **all**.

flag	meaning
`--unwind 1`	number of times to unwind loops
`--flush`	print to stdout
`--object-bits 8`	number of pointer bits to store meta information
`--malloc-may-fail`	malloc may fail
`--malloc-fail-null`	malloc may fail and return `NULL`

TABLE V: CBMC options for **no-mem-safe**.

flag	meaning
`--unwind 1`	number of times to unwind loops
`--flush`	print to stdout
`--object-bits 8`	number of pointer bits to store meta information
`--malloc-may-fail`	malloc may fail
`--malloc-fail-null`	malloc may fail and return `NULL`

TABLE VI: CBMC options **no-memmove**.

APPENDIX

Performance of SEABMC vs CBMC In this section we look at performance of SEABMC vs CBMC more closely. In App. A we study tool performance on a single task by using different features of the tools. In App. B, we look at the CBMC flags used for the analysis.

A. Comprehensive Analysis w.r.t. CBMC

SEABMC outperforms CBMC on many of the categories. To ensure that the comparison is "fair", we have done a comprehensive manual analysis with a few verification tasks.

For a fair comparison, one must show that the verification problem being solved is the same. While both tools verify user-supplied assertions in `aws-c-common`, they also verify internal properties such as memory safety, integer overflow, etc., depending on how they are invoked. For example, CBMC checks for integer overflow, while SEABMC does not. Hence, as a first step, we identified all such options in CBMC and disabled them.

There are many other factors that differentiate SEABMC and CBMC including: IRs (i.e., GOTO program vs. LLVM-IR), model of memory operations, and VCGen. Thus, we identified the differences that benefit SEABMC. We chose one verification task `aws-array-list-erase`, and derived 3 configurations based on the above analysis[11]: 1) *All*: SEABMC and CBMC verify a similar set of properties, namely, user-supplied assertions and memory safety. 2) *No Memory Safety*: SEABMC and CBMC verify user-supplied assertions only. 3) *No* **memmove**: `aws-array-list-erase` uses **memmove** in its implementation. Since **memmove** has custom implementations in both SEABMC and CBMC, we evaluated run-time when disabling the assertions for it.[12]

The results are shown in Tab. III. We present the analysis for one verification task, however, the same applied to other verification tasks where SEABMC outperforms CBMC– even when verifying similar properties. Further manual analysis shows that most difference is due to the model of memory in SEABMC and CBMC. Specifically, memory operations on large blocks, are very expensive for CBMC (40s vs. 98s due to pre-conditions for **memmove** in Tab. III).

B. Command line options for CBMC

This section lists the CBMC command line flags used for `aws-array-list-erase` verification job for different configuration.

all Options to enable user assertions and memory safety checks
no-mem-safe Options to enable user assertions only
no-memmove Options to enable user assertions and remove memory safety and **memmove** checks.
The **memmove** checks are disabled manually in source code.

[11] See App. B for CBMC flags used.

[12] These assertions guarantee spatial memory safety of **memmove**.