Bounded Model Checking for LLVM

Siddharth Priya∗
siddharth.priya@uwaterloo.ca
Xiang Zhou∗
x245zhou@uwaterloo.ca
Yusen Su∗
y256su@uwaterloo.ca
Yuyan Bao∗
yuyan.bao@uwaterloo.ca

Arie Gurfinkel†
arie.gurfinkel@uwaterloo.ca

∗University of Waterloo
Waterloo, Canada
†The Technion
Haifa, Israel

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 if $P$ 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-
then describe a series of transformations that transform a LLVM bitcode with purified memory operations. We 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.

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 $\text{nd\_char()}$. The value of $x$ is further constrained by the $\text{assume}$, 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 $\text{assert}$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 $\text{main}$, which consists of four basic blocks labeled by BB0, BB1, BB2 and BB3. A basic

\footnote{ITE stands for If-Then-Else.}
A program is in a Pure Dataflow Form. A (loop-free) program is in a Pure Dataflow Form if every definition appears on a def-use chain of either assume or assert, and contains a single definition of registers. Each such definition is said to be in the cone of influence (COI). In Fig. 4a, the highlighted code is diverted to ERR, 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. For example, \( \phi[M1, BB1], [M2, BB2] \) in ERR of Fig. 3c is transformed into select R2, M1, M2 in Fig. 3d, where 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 \( \text{br} \) 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,

2In LLVM, select is the usual ternary ITE such as \( a ? c : b \) in C.
Verification Condition Generation. We now describe the logic equation. Now, generating VC implies mapping each definition into a used in the encoding: bit-vector of of presentation, we assume that two fundamental sorts are

```plaintext
(a) SEA-IR
(b) Single Assert (SA)
(c) Single Assume (SASA)
(d) Gated SSA (GSSA)
```

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

```plaintext
(a) Pure-Dataflow (PD)
(b) SMT-LIB
```

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

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.

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, \( bv(64) \to bv(64) \).

In addition, we use the following helper sorts: \( scalr : bv(64) \), \( ptsr : scalr \), and \( mems : bv(64) \to bv(64) \), where \( scalr \) is sorts of scalars, \( ptsr \) 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 \( sym(r) \) 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, \( sym(R0) \) is \( r0 \) of \( scalr \) sort, \( sym(R2) \) is \( 2 \) of \( ptsr \) sort, and \( sym(M0) \) is \( m0 \) of \( mems \) sort, respectively.

Second, each expression \( E \) in SEA-IR is mapped into a corresponding SMT-LIB expression \( sym(E) \). 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 (\( ptof \)) 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 \( sym(E) \). For example, in Fig. 4a, \( A = select \ R5,R6,R1 \) is translated into \( a = ite(r5,r6,r1) \) in Fig. 4b.

Translating \( alloca \) and \( malloc \) requires a memory allocator. We parameterize \( sym \) by an allocation function \( alloca : A \to ptsr \) that maps allocation expressions in \( A \) to values of pointer sort. For example, in Fig. 5, \( P1, M1 = alloca R0, M0 \) is translated into \( p1 = alloca(alloca R0,M0) \land m1 = m0 \), and is reduced to \( p1 = addro \land m1 = m0 \), where \( addro \) is the return value of \( alloca \).

\(^3\)In practice, \( SEARMC \) supports multiple bit-widths for scalars, and different ranges for values for maps.
Memories are modeled by an SMT-LIB theory of
new memory obtained by writing the value
into
read
ister
m
read
function
load

guaranteed to be disjoint since Fig. 4b adds a constraint that
presentation of memories (see Sec. III). We use two functions,
easily added.

such as separating kernel- and user-space addresses can be

the memory register
M


∀ 
(a ∈  A · size(a)) is known
∀a ∈ A · (alloc(a) mod align(a)) = 0
∀a1 ≠ a2 ∈ A · (alloc(a1) + size(a1) ≤ alloc(a2)) ∨ (alloc(a2) + size(a2) ≤ alloc(a1))

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

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
p1 = addr0 ∧ p2 = addr0 + 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 rep-
resentation 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 reg-
ister m at index p. The function write(m,r1,p2) represents a
new memory obtained by writing the value r1 at index p2 in
m. In Fig. 5, load P0, M and store R1, P2, M0 are translated into
read(m,p0), and write(m0,r1,p2).

SEA/BMC has two memory representations: Arrays and
Lambdas.

Arrays. Memories are modeled by an SMT-LIB theory of
extensional arrays ArraysEx4. 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
λx.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,r0) is translated
into a function application m(r0), and write(m0,r1,p2) is
translated into a new λ-function, λx.ite(x = p2,r1,m0). In
the final VC, function applications are β-reduced to substitute
formal arguments with actual parameters. Thus, the VC only


\[
\begin{array}{|c|c|c|}
\hline
\text{read}(m,p_0) & \text{Array} & \lambda \\
\text{write}(m_0, r_1, p_2) & \text{store} m_0 r_1 p_2 & \lambda x.\text{ite}(x = p_2, r_1, m_0(x)) \\
\hline
\end{array}
\]

Fig. 7: Translation of read and write.

RDEF ::= R = isderef R | R = isalloc R.M | R = ismod R. M
Fig. 8: SEA-IR syntax for memory safety.

has ites, and does not require ArrayEx support in the SMT-
solver.

Overall, for a program P in a reduced PD form with a
sequence of statements s0, . . . , sk, followed by assume r0 and
assert r1, sym(P) is defined as follows:

\[
sym(P) \triangleq \bigwedge_{0 \leq i \leq k} \text{sym}(s_i) \land \text{sym}(r_0) \land \text{sym}(r_1).
\]

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 ¬false. The VC
is unsatisfiable since λ evaluates to false.

Theorem 1: sym(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 r, sym(P) is a tuple (t1, . . . , tj) of j
constants that represents the pointer, and addr((t1, . . . , tj)) 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((t1, . . . , tj)) = t1.

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
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 ⟨s1, . . . , sk⟩. We say that a tuple τ = ⟨c1, . . . , cp⟩ of p
constants is of a tuple sort ⟨s1, . . . , sp⟩ iff, for each 0 < i ≤ p,
ci is of sort si. Tuples of sorts, and tuples of constants are
only present during VCGen, but not in the final verification
fun main() {
BB0:  
M0 = mem.init()
R0 = nd_char()  
R1 = R0 > 0 ∧ R0 < 10  
R2 = ndbool()  
P1, M1 = malloc 2, M0  
P2, M2 = malloc 1, M0  
M3 = select P2, M1, M0  
P3 = select R2, P1, P2  
R4 = isderef R3, 1  
R5 = alloc R0, R4, R5
R6 = isderef R5, 1
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
}  
(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.

\[
sym\{P1,M1 = malloc R0,M0\} \triangleq\]
\[
\begin{align*}
p1 &= alloc(malloc R0, M0) ∧ m1 = alloc_sh(m1,p1) \\
\text{sym}(M1 = free P0,M0) \triangleq m1 = free_sh(m0,p0) \\
\text{sym}(Rr = store R1,P2,M2) \triangleq \\
\langle m_{r,1}, \ldots, m_{r,j} \rangle &= (\text{write}(m_{r,1}, r1, addr(p2)), \ldots, \text{write}(m_{r,j}, r1, addr(p2))) ∧ \\
\langle m_{j+1,1}, \ldots, m_{j,k} \rangle &= \text{store}_{sh}(\langle m_{j+1,1}, \ldots, m_{j,k} \rangle, p2) \\
\text{sym}(R1 = load P0,M0) \triangleq r1 &= (\text{read}(m_{0,1}, addr(p0)), \ldots, \text{read}(m_{0,j}, addr(p0)))
\end{align*}
\]

Fig. 10: Memory-safety aware VCGen semantics.

condition. For that, we rewrite equality between two tuples as conjunction of equalities between their elements, and use \( \tau.i \) for the \( i \)th element of tuple \( \tau \).

Similarly, we re-define \( \text{mems} \) for a memory register \( M \) to be a tuple of values that store the program and the shadow states. Thus, \( \text{sym}(M) = (v_0, \ldots, v_k) \), 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 \( \text{malloc} \) 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 \( \text{alloc} \) is unchanged, but now returns a fat pointer. Given a pointer \( p \) of sort \( \text{ptrs} \), a function \( \text{size}((t_1, \ldots, t_j)) \) returns the size of a memory object pointed-to by \( p \). An additional function \( \text{alloc}_{sh} : \text{mems} \rightarrow \text{mems} \) operates on shadow memory. The semantics of \( \text{alloc}_{sh} \) and \( \text{free}_{sh} \) is described later.

A \( \text{store} \) is divided into two parts. First is the store of the actual program data. Since the data can be of sort \( \text{scalar} \) or \( \text{ptrs} \), a store of a \( k \)-tuple of data on memory \( m0 \) is translated into \( k \) writes, on each element of \( \langle m_{0,1}, \ldots, m_{0,j} \rangle \). Second is updating metadata, done by \( \text{store}_{sh} \) that works on \( \langle m_{0,j+1}, \ldots, m_{0,k} \rangle \). The details of \( \text{store}_{sh} \) are described later in this section. Similarly, a \( \text{load} \) expects to read \( \langle m_{1,1}, \ldots, m_{1,j} \rangle \) of sort \( \text{ptrs} \).

Spatial memory safety A program satisfies spatial memory safety if every read and write is always inside an allocated object. A fat pointer is defined as a tuple of three constants \( (a_1, a_2, a_3) \) denoted as \( (\text{base}, \text{offset}, \text{size}) \) for convenience. Here \( \text{base} \) is the start address of the object, \( \text{offset} \) is an index into the object, and \( \text{size} \) is its size. The address \( \text{addr} \) is given by \( \text{base} + \text{offset} \).

With fat pointers, we introduce instructions for pointer arithmetic and pointer integer casts. The instruction \( \text{gep} \) is used for pointer arithmetic. Fig. 9a shows an example use in \( R5 = \text{gep} R3, 1 \). Here, semantically, a new pointer \( R5 \) is created that has the same \( \text{base} \) and \( \text{size} \) as \( R2 \), with \( \text{offset} \) incremented by 1. We also introduce \( \text{ptoi} \) instruction that casts a pointer to an integer by adding \( \text{offset} \) to \( \text{base} \). For an integer to pointer cast, we use the \( \text{itop} \) instruction. This instruction sets \( \text{base} \) to the integer value and \( \text{fat} \) (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 \( \text{assert}(0) \) as \( R6 = \text{isderef} R5, 1 \) evaluates to \( \text{false} \) causing \( A1 \) and \( A2 \) to evaluate to \( R1 \) and \( \text{true} \), 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 SBAMC, we automatically add isderef assertions before memory accesses. Many of such assertions are statically and, thus, cheaply resolved to \( \text{true} \) or \( \text{false} \) 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 invalid. 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
A program satisfies temporal memory safety iff it never does one of the following: (UA) an object is used after it has been freed; and (RO) an object marked as read-only (by programmers) is modified. We detect an object violation of memory safety by tracking the status of a memory object using shadow memory. Each memory is a tuple \((\text{val}, \text{offset}, \text{size}, \text{alloc}, \text{mod})\), where \((\text{val}, \text{offset}, \text{size})\) maps to pointer data \((\text{base}, \text{offset}, \text{size})\), and \text{alloc} and \text{mod} track the allocated and modified status of an object, respectively.

An object can be in allocated or freed state. To track allocated state, \text{sym} in Sec. II is extended for \text{alloca}, \text{malloc}, and \text{free}. The new semantics is shown in Fig. 10. The function \(\text{alloca} : \text{mems} \rightarrow \text{mems}\) is defined, for temporal memory safety, as shown in Fig. 11. Note that \(\text{alloca}(m, r)\) marks \(m.\text{mod}\) memory only at the start of an object, i.e., \(r.\text{base}\). For this reason it is necessary to use the fat pointer representation since it records the \text{base} for every pointer. The \text{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 \text{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 \text{store} (see Fig. 10). The function \(\text{store} : \text{mems} \rightarrow \text{mems}\) is implemented for temporal memory safety (see Fig. 11). The \text{ismod} in Fig. 8 is used to check the read-only state of an object at any program point. The semantics for \text{ismod} is given in Fig. 12. We also provide a companion instruction \text{resetmod} \(R, M\) that resets \(m.\text{mod}\) at address \(r.\text{base}\) 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-\text{c-common}. Each task verified post-conditions and memory safety of a single function from aws-\text{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 \textit{kinds} of verification tools on \textit{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 \textit{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. 1a. 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. 1b. Here, \textit{optimal} means applying all of the transformation involved, plus eagerly simplifying VC during VCGen. \(\beta\)-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) \text{rel} \_\text{alloc} – use \text{alloc} that returns relative addresses from some symbolic start of stack and heap, rather than concrete addresses 2) \text{flat} \_\text{mem} – one flat memory instead of using alias analysis to partition memory into disjoint memories as much as possible 3) \text{no} \_\text{coi} – disable cone-of-influence 4) \text{no} \_\text{simp} – disable eager simplification 5) \text{p} \_\text{cond} – generate VC directly from SSA form by using path condition to encode \text{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 (\textit{bv}(8)), 4 bytes (\textit{bv}(32)) and 8 bytes (\textit{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. 1c. 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 \textit{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


\(^6\)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.
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. I). This 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.

\[\text{LLBMC} \text{ source code is not publicly available; Binary download on website is broken.}\]

\[\text{Details at https://github.com/seahorn/verify-c-common/issues/124}\]

---

**TABLE I: Evaluations of different configuration.**

<table>
<thead>
<tr>
<th>Word size</th>
<th>Unsat</th>
<th>Timeout</th>
<th>Failed</th>
<th>Solved time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>coi</code></td>
<td>170</td>
<td>2</td>
<td>849</td>
<td>510</td>
</tr>
<tr>
<td><code>mem</code></td>
<td>163</td>
<td>9</td>
<td>2689</td>
<td>55</td>
</tr>
<tr>
<td><code>yices2</code></td>
<td>170</td>
<td>2</td>
<td>849</td>
<td>510</td>
</tr>
<tr>
<td><code>yices2</code></td>
<td>170</td>
<td>0</td>
<td>912</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 14: Incorrect byte_buf invariant**

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 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 maintaining 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
performance improvements over the competition. We have evaluated SEA and Symbiotic. Our representation shares SEA and KLEE that we compare with. Unfortunately, there is no public version of LLBMC available, which makes it difficult to compare against CBMC, a new 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 SEAMB. DIVE4 [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 SEAMB does not. For sequential programs, it is related to libFuzzer and KLEE that we compare with.

VI. CONCLUSION

We have presented the techniques behind SEAMB, a new LLVM-base Bounded Model Checker for C. SEAMB 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 SEAMB against CBMC, SMACK, SYMBIOTIC, and KLEE and show significant performance improvements over the competition.

REFERENCES


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

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>--unwind 1</td>
<td>number of times to unwind loops</td>
</tr>
<tr>
<td>--flush</td>
<td>print to stdout</td>
</tr>
<tr>
<td>--object-bits 8</td>
<td>number of pointer bits to store meta information</td>
</tr>
<tr>
<td>--malloc-may-fail</td>
<td>malloc may fail</td>
</tr>
<tr>
<td>--malloc-fail-null</td>
<td>malloc may fail and return NULL</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>verification task</th>
<th>config</th>
<th>tool</th>
<th>run-time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws-array-list-erase</td>
<td>all</td>
<td>SEABMC</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>no-mem-safe</td>
<td>SEABMC</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>no-memmove</td>
<td>SEABMC</td>
<td>3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>--unwind 1</td>
<td>number of times to unwind loops</td>
</tr>
<tr>
<td>--flush</td>
<td>print to stdout</td>
</tr>
<tr>
<td>--object-bits 8</td>
<td>number of pointer bits to store meta information</td>
</tr>
<tr>
<td>--malloc-may-fail</td>
<td>malloc may fail</td>
</tr>
<tr>
<td>--malloc-fail-null</td>
<td>malloc may fail and return NULL</td>
</tr>
<tr>
<td>--bounds_check</td>
<td>check access is within bounds</td>
</tr>
<tr>
<td>--pointer_check</td>
<td>check access is within bounds</td>
</tr>
</tbody>
</table>

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

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:

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.

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.