An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation

Juneyoung Lee[1], Dongjoo Kim[1], Chung-Kil Hur[1], and Nuno P. Lopes[2]

[1] Seoul National University
[2] Microsoft Research

Abstract. Several automatic verification tools have been recently developed to verify subsets of LLVM's optimizations. However, none of these tools has robust support to verify memory optimizations.
In this paper, we present the first SMT encoding of LLVM's memory model that 1) is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations, and 2) enables bounded translation validation of programs with up to hundreds of thousands of lines of code. We implemented our new encoding in Alive2, a bounded translation validation tool, and used it to uncover 21 new bugs in LLVM memory optimizations, 10 of which have been already fixed. We also found several inconsistencies in LLVM IR's official specification document (LangRef) and fixed LLVM's code and the document so they are in agreement.

1 Introduction

Ensuring that LLVM is correct is crucial for the safety and reliability of the software ecosystem. There has been significant work towards this goal including, e.g., formally specifying the semantics of the LLVM IR (intermediate representation). This entails describing precisely what each instruction does and how it handles special cases such as integer overflows, division by zero, or dereferencing out-of-bounds pointers [8, 24, 26, 29, 47]. There has also been work on automatic verification of classes of optimizations, such as peephole optimizations [25, 31], semi-automated proofs [48], translation validation [20, 35, 42, 44], and fuzzing [23, 46]. All this work uncovered several hundred bugs in LLVM.

While there has been great success in improving correctness of scalar optimizations, current verification tools only support basic memory optimizations, if any. Since memory operations can take a significant fraction of a program's run time, memory optimizations are very important for performance. The implementation of these optimizations and related pointer analyses tends to be complex, which further justifies the investment in verifying them.

Verifying programs with memory operations is very challenging and it is hard to scale automatic verification tools that handle these. The main issue lies with pointer aliasing: which objects does a given memory operation access? Without any prior information, a verifier must consider that each operation *may* load or store from any live object (global variables and stack/heap allocations). This creates a big case split for the underlying constraint solver to (attempt to) solve.

Since automatic verification of the source code of memory optimizations is out of reach at the moment, we focus on bounded translation validation [30, 40] (BTV) instead. (Bounded) translation validation consists in verifying that an optimization was correct for a particular input program (up to a bounded unrolling of loops) rather than verifying its correctness for all input programs.

In this paper, we present the first SMT encoding of LLVM's memory model [24] that is precise enough to validate all of LLVM's intraprocedural memory optimizations. The design of the encoding was guided by practical insights of the common aliasing cases in BTV to achieve better performance. For example, we observed that in most cases we can cheaply infer whether a pointer aliases with a locally-allocated or a global object (but not both). Therefore, our encoding case-splits itself on this property rather than leaving that to the SMT solver, as we can cheaply resolve the case split for over 95% of the cases.

The second contribution of this paper is a new semantics for heap allocation for the verification of optimizations for real-world C/C++ programs. Although LLVM's memory model has a reasonable semantics for heap allocations [24], we realized it was not suitable for verifying optimizations. In some programming styles, the result of functions such as `malloc` is not checked against `NULL` and the resulting pointer is dereferenced right away. Since `malloc` can return `NULL` in some executions, we could end up proving that some undesirable optimizations were correct since the program triggers undefined behavior in at least one execution. We propose a new semantics for heap allocations in this paper that is better suited for the verification of optimizations.

The third contribution is the identification of approximations to the SMT encoding such that it is still sufficiently precise to verify (and find bugs) in LLVM's memory optimizations. This is possible since for translation validation we only need to be as precise as LLVM's static analyses (e.g., in the encoding of aliasing rules), and therefore we do not need to consider extremely precise analyses nor arbitrary transformations. Compilers have limited reasoning power by construction in order to keep compilation time reasonable.

We implemented our new SMT encoding of LLVM's memory model in Alive2 [30], a bounded translation validation tool for LLVM. We used Alive2 to find and report 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

To summarize, the contributions of this paper are as follows.

1. The first SMT encoding of LLVM's memory model that is precise enough to verify all of LLVM's intraprocedural memory optimizations.
2. A new semantics for heap allocations for the verification of optimizations of real-world C/C++ programs (§5.1).
3. A set of approximations to the SMT encoding to further improve the performance of verification without introducing false positives or false negatives in practice (§9).
4. Thorough evaluation of LLVM's memory model against LLVM's implementation, which uncovered deviations from the model (§10.3).
5. Identification of 21 previously unknown bugs in LLVM. We present a few examples in §10.1.

2 Overview

Consider the functions below in C:[3] a source (original) function on the left and a target (optimized) function on the right. According to the semantics of high-level languages, and also of LLVM IR, a pointer received as argument or a callee cannot guess the address of a memory region allocated within a function. That is, pointer `q` is not aliased with `p`, `r`, nor touched by `g(p+1)`. Although the caller of `f` may guess the address of `q` in practice, that behavior is excluded by the language semantics because `p`'s object (*provenance*) cannot be a fresh one like `q`. If `p` happens to alias `q`, accessing such pointer triggers undefined behavior (UB).

```
1  int f(int *p) {
2    int *q = malloc(4);
3    *q = 42;
4    int *r = g(p+1);
5    *r = 37;
6    return *q;
7  }
```

```
1′ int f(int *p) {
2′   // q removed
3′
4′   int *r = g(p+1);
5′   *r = 37;
6′   return 42;
7′ }
```

The provenance rules allow LLVM to forward the stored value in line 3 to line 6, and therefore line 6′ simply returns 42. As the value stored to `*q` is not used anymore and pointer `q` does not escape, LLVM also removes the heap allocation.

Next we show how to verify this example. Note that we do not require the two programs to be aligned; the example is aligned to make it easier to understand.

2.1 Verifying The Example Transformation

We start by defining two auxiliary functions that encode the effect of memory operations on the program state. Let state $S = (m, ub)$ be a pair, where m is a memory and ub a boolean that tracks whether the program has already executed UB or not. Let p be the accessed pointer, and v the stored value. The definition of functions $\overline{\textbf{load}}$ and $\overline{\textbf{store}}$ is as follows:

$$\overline{\textbf{load}}\ p\ S ::= (\ \textbf{load}(p, S.\mathsf{m})\ ,\ (S.\mathsf{m},\ S.\mathsf{ub} \vee \neg\ \textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})\ ))$$
$$\overline{\textbf{store}}\ p\ v\ S ::= (\ \textbf{store}(p, v, S.\mathsf{m})\ ,\ S.\mathsf{ub} \vee \neg\ \textbf{deref}(p, \texttt{sizeof}(*p), S.\mathsf{m})\ )$$

$\overline{\textbf{load}}$ returns a pair with the loaded value and the updated state, where ub is further constrained to ensure that pointer p is dereferenceable for at least the size of the loaded type. Similarly, $\overline{\textbf{store}}$ returns the updated state. The gray boxes ($\cdots$) encode SMT expressions; we describe these in the next section.

1. Encoding the output states. Table 1 shows the state after executing each of the programs' lines. p, m_0, and ub_0 are SMT variables for the input pointer, and function `f` caller's memory and UB flag, respectively. The target's corresponding variables are primed. Meta variables are upper-cased and SMT variables are lower-cased.

[3] We use the syntax of C for many of the examples in this paper to make them easier to read, even though we consider the semantics of LLVM IR.

#	**Inputs:** p, m_0, ub_0	#	**Inputs:** p', m_0', ub_0'
2	$S_1 := (m_0, ub_0)$ $\quad A_1 :=$ q is fresh	$2'$	-
3	$S_2 := \overline{\textbf{store}}\ q\ 42\ S_1$	$3'$	-
4	$S_3 := (m_\mathtt{g}, S_2.\mathsf{ub} \vee ub_\mathtt{g})$ $A_2 :=$ r is not aliased with $q \wedge$ $m_\mathtt{g}$ agrees with $S_2.\mathsf{m}$ on q	$4'$	$S_1' := (m_\mathtt{g}', ub_0' \vee ub_\mathtt{g}')$
5	$S_4 := \overline{\textbf{store}}\ r\ 37\ S_3$	$5'$	$S_2' := \overline{\textbf{store}}\ r'\ 37\ S_1'$
6	$\mathrm{O} := \overline{\textbf{load}}\ q\ S_4$	$6'$	$\mathrm{O}' := (42, S_2')$

Table 1. States and axioms after executing each of the lines of `f`.

On line 2, q is assigned a pointer to a new object (encoded in axiom A_1). On line 3, '`*q = 42`' updates the state using $\overline{\textbf{store}}$.

On line 4, the return value, output memory, and UB of `g(p+1)` are represented with fresh variables r, $m_\mathtt{g}$, and $ub_\mathtt{g}$, respectively. Axiom A_2 encodes the provenance rules: the return value cannot alias with locally non-escaped pointers (`q`) and only the remaining objects are modified. Line $4'$ does not need these axioms because there are no locally-allocated objects in the target function.

Finally, the outputs O and O' are a pair of return value and state.

2. Relating the source and target's states. To prove correctness of a transformation, we must first establish refinement between the input states of the source/target functions. Refinement ($\sqsupseteq$) is used rather than equality because it is allowed for the source's caller to give less defined inputs than the target's.

$$A_{\text{in}} := p \sqsupseteq p' \wedge m_0 \sqsupseteq m_0' \wedge (ub_0' \implies ub_0)$$

The inputs and outputs of function calls are also related using refinement. For any pair of calls in the source and target functions, if the target's inputs refine those of the source, the target's output also refines the source's output. The example only has one function call pair:

$$A_{\text{call}} := \Big(S_2.\mathsf{m} \sqsupseteq m_0' \wedge p+1 \sqsupseteq p'+1 \implies m_\mathtt{g} \sqsupseteq m_\mathtt{g}' \wedge r \sqsupseteq r' \wedge (ub_\mathtt{g}' \implies ub_\mathtt{g}) \Big)$$

We can now state the correctness theorem for the example transformation. For any input, if the axioms hold, the output of the target must refine that of the source for some internal nondeterminism in the source (e.g., the address of pointer `q`). Output is refined iff (i) the source triggers UB, or (ii) the target triggers no UB, and the target's return value and memory refine those of the source.

$$\forall p, p', m_0, m_0', ub_0, ub_0', m_\mathtt{g}, m_\mathtt{g}', ub_\mathtt{g}, ub_\mathtt{g}' .\ \exists q .\ (A_1 \wedge A_2 \wedge A_{\text{in}} \wedge A_{\text{call}}) \implies O \sqsupseteq O'$$

2.2 Efficiently Encoding LLVM's Memory Model and Refinement

We now present our key ideas for efficiently encoding LLVM's memory model and refinement (the gray boxes) in SMT, which is one of our main contributions.

1. Pointers. We represent a pointer as a pair (bid, o) of a block id (i.e., its provenance) and an offset within, so that we can easily detect out-of-bound

accesses: accessing (bid, o) in memory m triggers UB unless $0 \leq o < m[bid].\text{size}$, from which $\textbf{deref}((bid, o), sz, m)$ naturally follows.

2. Bounding the number of blocks. Our first observation is that we can safely bound the number of memory blocks for *bounded* translation validation since loops are unrolled for a fixed number of iterations. As a result, we can use a (fixed-length) bit-vector to encode block ids.

For the example source function, four blocks are sufficient: three for pointers `p`, `q`, `r` as they may all point to different blocks, and an extra to represent all the other blocks that are not syntactically present but are accessible by function `g`.

For the sake of simplifying the example, we ignore that `p`, `q`, `r` may be **null**. Our model does not make such assumption; we explain later how null is handled.

3. Aliasing rules. Several of the aliasing rules are encoded for free as we can distinguish most blocks by construction. First, we use the most significant bit of the block ids to distinguish local (1) from non-local (0) blocks. Second, we assign constant ids whenever possible (e.g., global variables and stack allocations).

For the example source function, (without loss of generality) we set the block ids of `q`, `p` and the extra block to $100_{(2)}$, $000_{(2)}$, and $011_{(2)}$ (in binary format), respectively. However, we cannot fix the block id of `r` and instead give the constraint that it should be either $000_{(2)}$ or $001_{(2)}$ since `r` may alias with `p` but not with `q`. This establishes the alias constraints in A_1 and A_2 for free.

4. Memory accesses. In order to leverage the fact that each pointer may range over a small number of blocks as seen above, we use one SMT array per block (from an offset to a byte) instead of using a single global array (from a pointer to a byte). For the latter, it becomes harder to exploit non-aliasing guarantees since all stores to different blocks are grouped together.

For the example source function, m_0 consists of four arrays $m_0^{(100)}$, $m_0^{(000)}$, $m_0^{(001)}$, $m_0^{(011)}$ for the four blocks. Then since q's block id is $100_{(2)}$, $\overline{\textbf{store}}\ q\ 42\ S_1$ at line 3 only updates the array $m_0^{(100)}$, leaving the others unchanged. Similarly, $\overline{\textbf{store}}\ r\ 2\ S_3$ at line 5 only updates $m_0^{(000)}$ and $m_0^{(001)}$ using the SMT if-then-else expression on r's block id. Finally, $\overline{\textbf{load}}\ q\ S_4$ at line 6 reads from the updated array at $100_{(2)}$, thereby easily realizing that the read value is 42.

5. Refinement. The value/memory refinement $\sqsupseteq$ is defined based on a mapping between source and target blocks, which we efficiently encode leveraging the alignment information between source and target as much as possible (§7).

3 LLVM's Memory Model

In this section, we give a brief introduction to LLVM's memory model [24]. In this paper we only consider logical pointers (i.e., integer-to-pointer casts are not supported) and a single address space.

Memory Block A memory block is the unit of memory allocation: each stack or global variable has a distinct block, and heap allocation functions like **malloc** create a fresh block each time they are called. Each block is uniquely identified

with a non-negative integer (bid), and has associated properties, including size, alignment, whether it can be written to, whether it is alive, allocation type (heap, stack, global), physical address, and value.

Pointer. A pointer is defined as a triple $(\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$, where off is an offset within the block bid, and attrs is a set of attributes that constrain dereferenceability and which operations are allowed.

Pointer arithmetic operations (**gep**) only change the offset, with bid and attrs being carried over. Unlike C, an offset is allowed to go out-of-bounds (OOB). Such pointer, however, cannot be dereferenced like in C (triggers undefined behavior—UB), but can be used for pointer comparisons for example.

LLVM supports several pointer attributes. For example, a **readonly** pointer p cannot be used to store data. However, it is possible to use a non-**readonly** pointer q to store data to the same location as p (provided the block is writable). A **nocapture** pointer cannot escape from a function. For example, when a function returns, no global variable may have a **nocapture** pointer stored (otherwise it is UB).

LLVM has three constant pointers. The **null** pointer is defined as $(0, 0, \emptyset)$. Block 0 is defined as zero sized and not alive. The **undef**[4] pointer is defined as $(\beta, \delta, \emptyset)$, with β, δ being fresh variables for each observation of the pointer. There is also a **poison**[5] pointer.

Instructions. We consider the following LLVM memory-related instructions:

- Memory access: **load**, **store**
- Memory allocation: **malloc**, **calloc**, **realloc**, **alloca** (stack allocation)
- Lifetime: **start_lifetime** (for stack blocks), **free** (stack/heap deallocation)
- Pointer-related: **gep** (pointer arithmetic), **icmp** (pointer comparison)
- Library functions: **memcpy**, **memset**, **memcmp**, **strlen**
- Others: **ptrtoint** (pointer-to-integer cast), **call** (function call).

Unsupported memory instructions are: integer-to-pointer casts, and atomic and volatile memory accesses.

4 Encoding Memory Blocks and Pointers in SMT

We describe our new encoding of LLVM's memory model in SMT over the next few sections. We use the theories of UFs (uninterpreted functions), BVs (bit-vectors), and arrays with lambdas [7], with first order quantification. Moreover, we consider that the scope of verification is a single function without loops (or where loops have been previously unrolled).

[4] In LLVM, **undef** values are arbitrary values of a given type with the additional property that they can yield a different value each time they are observed. **undef** values can be replaced with any value of the same type, except **poison** values.

[5] A **poison** value taints whole expression trees (e.g., **poison** + 1 = **poison**), and branching on it is UB. Similarly, dereferencing a **poison** pointer is UB.

4.1 Memory Blocks

Each memory block is assigned a distinct identifier (a bit-vector number). We further split memory blocks into local and non-local. Local blocks are all those that are allocated within the function under consideration, either on the stack or the heap. Non-local blocks are the remaining ones, including global variables, heap/stack allocations in callers and heap allocations in callees (stack allocations in callees are not observable, since they are deallocated when the called function returns, hence there is no need to consider them).

We use the most significant bit (MSB) to encode whether a block is local (1) or non-local (0). This representation allows the null block to have $\mathsf{bid} = 0$ and be non-local. We refer to the short block id, or $\widetilde{\mathsf{bid}}$, to refer to bid without the MSB. This is used in cases where it has already been established whether the block is local or not. Example with 4-bit block ids:

```
int g;                 // bid(g) = 0001
void f(int *p) {       // bid(p) = 0xyz (with xyz = arbitrary)
  int a[2];            // bid(a) = 1000
  int *q = malloc(4);  // bid(q) = 1001
}
```

The separation of local and non-local block ids is an efficient way to encode the constraint that pointers of these groups cannot alias with each other. In the example above, argument `p` cannot alias with either `a` or `q`.

As we only consider functions without loops, block ids can be statically assigned for each allocation site.

4.2 Pointers

A pointer $\mathsf{ptr} = (\mathsf{bid}, \mathsf{off}, \mathsf{attrs})$ is encoded as a single bit-vector consisting in the concatenation of the three elements. The offset is interpreted as a *signed* number (which is why blocks cannot be larger than half of the address space). Each attribute (such as **readonly**) is encoded with a bit. Example with 2-bit block ids and offsets, and a single attribute (we use `.` to visually separate the elements):

```
void f(char readonly *p, char *q) { // p = 0x.ab.1, q = 0y.cd.0
  char *r = p + 2;                  // r = 0x.(ab+2).1
  char *s = q + 3;                  // s = 0y.(cd+3).0
  char *t = malloc(4);              // t = 10.00.0
}
```

Let $\widetilde{\mathsf{off}}$ be a truncated offset where the least significant bits corresponding to the greatest common divisor of the alignment and sizes of all memory operations are removed. For example, if all operations are 4-byte aligned and they access either 4- or 8-byte values, then $\widetilde{\mathsf{off}}$ has less 2 bits than off (as these are guaranteed to be always zero when accessing the memory).

4.3 Block Properties

Each block has seven associated properties: size, alignment, read-only, liveness, allocation type (heap, stack, global), physical address, and value. Block properties are looked up and updated by memory operations. For example, when doing a store, we need to check if the access is within the bounds of the block.

Except for liveness and value, properties are fixed at allocation time. Liveness is encoded with a bit-vector (one bit per block), and value with arrays (indexed on $\widetilde{\mathsf{off}}$). We use a multi-memory encoding, where we have one array per bid.

The encoding of fixed properties differs for local and non-local blocks. For non-local blocks, we use a UF symbol per property, taking $\widetilde{\mathsf{bid}}$ as argument. For local blocks, we cannot use UFs because for the refinement check some of these would have to be quantified (c.f. §7) and most, if not all, SMT solvers do not support quantification of UF symbols. Therefore, we encode each of the remaining properties of local blocks as an if-then-else (ITE) expression, which is tailored for each use (e.g., each time an operation needs to lookup a local block's size, we build an ITE expression for the given $\widetilde{\mathsf{bid}}$).

Using ITE expressions to encode properties is less concise than using UFs. However, it is not a disaster for two reasons. Firstly, we only need to consider the local blocks that have been allocated beforehand, since the program cannot access blocks allocated afterward. Secondly, pointers are usually not fully arbitrary. Oftentimes we know statically which type of block they refer to, and even what is the block id, given that pointer arithmetic operations do not change the block id. Therefore, the ITE expressions are usually small in practice. Example with 4-bit block ids and offsets of a source program:

```
int g;                          // g = 0001.0000, size_src(001) = 4
void f() {
  char p[2];                    // p = 1000.0000
  char q[3];                    // q = 1001.0000
  char *r = ... p or q or g ...
  r[2] = 0;
  char t[1];                     // t = 1010.0000
}
```

The store in this program is only well defined if the size of block pointed by `r` is greater than 2. This is encoded in SMT as follows:

$$\mathsf{ite}(\mathbf{islocal}(r), \mathsf{ite}(\widetilde{\mathbf{bid}}(r) = 0, 2, 3), \mathsf{size}_{src}(\widetilde{\mathbf{bid}}(r))) > 2$$

Function $\mathbf{islocal}(p)$ is encoded with the SMT $\mathsf{extract}$ expression to fetch the MSB of the pointer. Similarly, $\widetilde{\mathbf{bid}}(p)$ extracts the relevant bits from a pointer. The expression for local blocks only needs to consider local blocks 0 and 1, since block 2 (`t`) is only allocated afterward. This allows a simple single pass through the code to generate optimized ITE expressions.

Value Value is defined as an array from short offset to byte (described later in §6.1). For non-local blocks, only those that are constant are initialized with

the respective value. The remaining blocks are allowed to take almost any value. The exception is for pointers: non-local blocks cannot initially have local pointers stored, since the calling environment cannot fabricate local pointers.

Local blocks are initialized with **poison** values using a constant array (i.e., an array that yields the same value for all indexes).

4.4 Physical Addresses

If a program observes addresses (through, e.g., pointer-to-integer casting), we need additional constraints to ensure that addresses of blocks that overlap in time are disjoint. Since we are doing translation validation, we have two programs with potentially different sets of locally allocated blocks. Therefore, we need to ensure that non-local blocks' addresses are disjoint from those of local blocks of both programs. This makes the disjointness constraints quite complex.

As an optimization, we split the address space in two: local blocks have MSB=1 and non-locals have MSB=0. Since the encoding of address disjointness is quadratic in the worst case (cross-product of blocks), halving the number of blocks is significant. This optimization, however, is an under-approximation of the program's behavior (§9). After investigating LLVM's optimizations, we believe it is highly unlikely this approximation will cause false negatives.

If a program does not observe any pointer's physical address, neither the block's physical address property nor the disjointness axioms are instantiated. However, when dereferencing a pointer, we need to check if the physical address is sufficiently aligned. When physical addresses are not created, we resort to checking alignment of both of the pointer's block and offset. Since in this case physical addresses are not observed (and therefore not constrained by the program using, e.g., pointer comparisons), a block's physical address can take any value, and therefore blocks and offsets must be both sufficiently aligned to ensure that physical pointers are aligned in all program executions. This argument justifies why we can soundly discard physical addresses.

4.5 Pointer Comparison

Given two pointers p and q, if a program learns that q is placed right after p in memory, the program can potentially change the contents of q without the compiler realizing it. Detecting the existence of such code is impossible in general, hence restricting the ways a program can learn the layout of objects in memory is important to make pointer analyses fast yet precise.

A way the memory layout can leak is through pointer comparison. For example, what should $\mathsf{p} < \mathsf{q}$ return if these point to different memory blocks? If it is a well-defined operation (i.e., simply compares their integer values), it leaks memory layout information. An alternative is to return a non-deterministic value to prevent layout leaks, the formal semantics of which is defined at [24].

We found that there are pros and cons of both semantics for the comparison of pointers of different blocks, and that neither of them covers all optimizations that LLVM performs. Table 2 summarizes the effects on each of the optimizations.

	Integer comparison	Non-deterministic
Fold $\mathsf{p} = \mathsf{q}$ to false if p.bid $\neq \mathsf{q}$.bid	No	Yes
Fold $\mathsf{p} + i = \mathsf{q} + i$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $(int)\mathsf{p} = (int)\mathsf{q}$ to $\mathsf{p} = \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{p} \neq \mathsf{q}$ to $\mathsf{p} < \mathsf{q}$	Yes	No
Fold $\mathsf{p} < \mathsf{q} \wedge \mathsf{q} \neq null$ to $\mathsf{p} < \mathsf{q}$	Yes	Potentially
Run-time aliasing checks	Yes	Correct, but not useful
Analysis of pointers cast from integers	Harder	Easy

Table 2. Comparison of two semantics for pointer comparison.

We decided to implement the integer comparison semantics, as LLVM performs all the optimizations above and its alias analyses (AA) mostly give up when they encounter an integer-to-pointer cast. In summary, we have to remove the first optimization from LLVM to make it sound. Additionally, we make it harder to improve LLVM's AA algorithms w.r.t. to pointers cast from integers.

4.6 Bounding the Maximum Number of Blocks

Since we assume that programs do not have loops, we can statically bound the maximum number of both local and non-local blocks a program may observe.

The maximum number of local blocks in the source and target programs, respectively, N^{src}_{local} and N^{tgt}_{local}, is computed by counting the number of heap and stack allocation instructions. Note that this is an upper-bound because not all allocation sites may be reachable in practice.

For non-local blocks, we cannot see their definitions as with local blocks, except for global variables. Nevertheless, we can still bound the maximum number of observed blocks. It is sufficient to count the number of instructions that may return non-local pointers, such as function calls and pointer loads. In addition, we consider a null block when needed (if the null pointer may be observed).

To encode the behavior of source and target programs, we need $N^{src}_{nonlocal} + N^{tgt}_{nonlocal}$ non-local blocks in the worst case, as all referenced pointers may be distinct. However, correct transformations will not have the target program observe more blocks than the source. If the target observes a pointer to a non-local block that was not observed in the source, we can set that pointer to **poison** because its value is not restricted by the source. Therefore, $N^{src}_{nonlocal}$ non-local blocks are sufficient to allow the target to exhibit *an* incorrect behavior.

The bit-width of $\widetilde{\mathsf{bid}}$ is: $w_{\widetilde{bid}} = \lceil \log_2(max(N^{src}_{nonlocal}, max(N^{src}_{local}, N^{tgt}_{local}))) \rceil$. When only local or non-local pointers are used, $w_{bid} = w_{\widetilde{bid}}$, as we know statically if the pointer is local or not. Otherwise, $w_{bid} = w_{\widetilde{bid}} + 1$.

5 Memory Allocation

In LLVM, memory blocks can be allocated on the stack (**alloca**), in the heap (e.g., **malloc**, **calloc**, etc), or as global variables. It is surprisingly non-trivial to find a semantics for memory allocations that allows all of LLVM's optimizations,

and rejects undesired transformations. For example, we have to support allocation removal and splitting, introduce new stack allocations and new constant global variables, etc. We explore multiple semantics and show their merits and shortcomings in the context of proving correctness of program transformations.

5.1 Heap Allocation

Heap allocation is done through functions such as **malloc**, **calloc**, C++'s `new` operator, etc. We describe semantics for **malloc**; remaining functions can be described in terms of it.

First of all, it is important to note that there are two common idioms used in practice by C programmers when doing memory allocation:

```
int *p = malloc(4);          |     int *p = malloc(4);
*p = 0;                      |     if (p) { *p = 0; }
```

In some programs, like the example on the left, **malloc** is assumed to never return **null**, say non-null assumption. This is mainly because the program does not consume too much memory and it is expected that the computer has enough memory/swap space. In other programs like the one on the right, **malloc** is expected to sometimes return **null**, say may-null assumption. Therefore, the program performs null-ness checks.

Since both programming styles are prevalent, we would like optimizations to be correct for both. This is non-trivial, as the two assumptions are conflicting: with the non-null assumption, it is sound to eliminate **null** checks, but not with the may-null assumption. We now explore several possible semantics to find one that works for both programming styles.

A. Malloc always succeeds. Based on the non-null assumption, in this semantics we only consider executions where there is enough space for all allocations to succeed. Regardless of whether the target uses more or less memory than the source, all calls to **malloc** yield non-null pointers. Therefore, for example, deleting unused **malloc** calls is allowed.

However, removing **null** checks of **malloc** is also allowed in this semantics. For example, optimizing the right example above into the left one is sound. This transformation, however, is obviously undesirable.

B. Malloc only succeeds if there is enough free space. To solve the problem just described, based on the may-null assumption, we can simulate the behavior of dynamic memory allocation and define **malloc** to return a pointer to a newly created block if there is an empty space in memory, and **null** otherwise. This semantics prevents the removal of **null** checks of **malloc** as it may return **null**.

However, this semantics does not explain removal of unused allocations. It aligns both source and target programs' allocations such that any change in the allocation sequence disrupts the program alignment and thus makes verification fail. For example, the following transformation removing unused **malloc** instructions and replacing comparisons of their output with **null** is not supported:

```
int *x = malloc(4);                ⇒    // remove x (unused)
if (x != nullptr) { ... }               if (true) { ... }
```

In case there were 0 bytes left in memory, `x` would be **null**, but since LLVM assumes that the program cannot observe the state of the allocator it folds the comparison `x != nullptr` to `true` after eliminating the allocation. This optimization would be flagged as incorrect in this semantics.

LLVM assumes very little about the run-time behavior of memory allocators. This is to support, for instance, garbage collectors, where an allocation may fail but if repeated it may succeed because memory was reclaimed in between. This explains why LLVM folds comparisons with **null** of unused memory blocks, and also contradicts the linear view of allocations of this semantics.

C. Malloc non-deterministically returns null. This semantics abstracts the behavior of the memory allocator by (1) allowing **malloc** to non-deterministically return **null** even if there is available space, and (2) only considering executions where there is enough space for all allocations to succeed. This semantics prevents the removal of null checks of **malloc**, which fixes the shortcomings of semantics A, and also allows the removal of unused allocations, which fixes those of semantics B. However, this semantics is too weak and therefore allows other undesirable transformations, like the following:

```
p = malloc(4);
*p = 0;                    ⇒        exit();
```

For the sake of proving refinement (§7), we need just one trace triggering UB (i.e., one particular realization of the non-deterministic choices) for a given input to be able to transform the source program into anything for that input. Informally speaking, refinement always picks the worst-case execution for each input. Since the source program executes UB when `p` is **null**, it is correct to transform the source into any program although that is obviously undesirable.

This semantics is too weak in practice since many programs are written without **null** checks, either assuming the program will not run out of memory, or assuming the program will terminate if it runs out memory. It is not reasonable in practice to allow compilers to break all such programs.

Our solution. As we have seen, there is no single semantics that both allows all desired transformations and rejects undesired ones. While semantics B prevents desired optimizations like allocation removal, semantics A and C allow undesired optimizations, but in a complementary way. For example, removing null checks of **malloc** is allowed in A but not in C. On the other hand, transforming an access of a **malloc**-allocated block without a **null** check beforehand into arbitrary code is allowed in C but not in A.

Therefore, we obtain a good semantics by requiring both A and C: an optimization is correct if it passes the refinement criteria with each of the two semantics. Intuitively, this definition requires the compiler to support the two considered coding styles: semantics A supports the non-null assumption, while semantics C the may-null assumption.

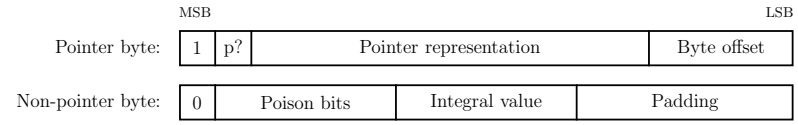

Fig. 1. Bit-wise representation of a byte. A pointer byte is poison if 'p?' is zero. A non-pointer byte tracks poison bit-wise.

5.2 Stack Allocation

The semantics of **alloca**, the stack-allocation instruction, is slightly different from that of **malloc**. LLVM assumes that stack allocations always succeed, since the program will likely crash if there is a stack overflow. That is, **alloca** never returns a **null** pointer.

LLVM performs more optimizations on stack allocations than on heap ones. For example, LLVM can split an allocation into multiple smaller ones or increase the alignment. These transformations can increase memory consumption.

6 Encoding Loads and Stores in SMT

We encode the value of memory blocks with several arrays (one per bid): from short offset to byte. We next give the definition of byte and the encoding of memory accessing instructions in SMT.

6.1 Byte

There are two types of bytes: *pointer* bytes and *non-pointer* bytes, cf. Fig. 1.

A **pointer byte** has the most significant bit (MSB) set to one. The following bit states whether the byte is poison or not. Next is the pointer representation as described in §4.2 ($\mathsf{bid}, \mathsf{off}, \mathsf{attrs}$).

Pointers are often longer than one byte, so when storing a pointer to memory we write multiple consecutive bytes. Each of these bytes records the same pointer, but with a different byte offset (the last bits of the byte) to distinguish between the partial bytes of the pointer.

For **non-pointer bytes**, we track whether each of the bits is poison or not. This is not required for pointers, since LLVM does not allow pointer values to be manipulated bit-wise. Non-pointer values can be manipulated bit-wise (e.g., using vectors with element types smaller than 8 bits). Each bit of the integral value is only significant if the corresponding poison bit is zero.

6.2 Load and Store Instructions

Load and store instructions are trivially encoded using SMT arrays. These arrays store bytes as described in the previous section. We next describe how LLVM values are encoded to and decoded from our byte representation.

We define two functions, $ty{\Downarrow}(v)$ and $ty{\Uparrow}(b)$, which convert a value v into a byte array and a byte array b back to value, respectively. We show below $ty{\Downarrow}(v)$

when $v \neq \textbf{poison}$. $\textbf{i}sz$ stands for the integer type with bit-width sz. If sz is not a multiple of 8 bits, v is zero-extended first. When v is poison, all poison bits are set to one. $\text{BitVec}(n, b)$ stands for number n with bit-width b. Pointer's byte offset is 3 bits because we assume 64-bit pointers.

$$\textbf{i}sz\Downarrow(v) \text{ or } \textbf{float}\Downarrow(v) = \lambda i.\, 0 +\!\!+ 0^8 +\!\!+ \text{bitrepr}(v)[8\times i \ldots 8\times(i+1)-1] +\!\!+ \text{padding}$$
$$ty*\Downarrow(v) = \lambda i.\, 1^2 +\!\!+ \text{bitrepr}(v) +\!\!+ \text{BitVec}(i, 3)$$

$\textbf{i}sz\Uparrow(b)$ and $\textbf{float}\Uparrow(b)$ return **poison** if any bit is **poison**, or if any of the bytes is a pointer. Otherwise, these functions return the concatenation of the integral values of the bytes.

$ty*\Uparrow(b)$ returns **poison** if any of the bytes is **poison** or not a pointer, there is more than one distinct pointer value in b, or one of the bytes has an incorrect byte offset (they have to be consecutive, from zero to byte size minus one). An exception is reading a non-pointer zero byte, which is interpreted as a null pointer byte. This allows initialization of, e.g., arrays with null pointers with **memset** (which is an idiom commonly used in LLVM IR).

6.3 Multi-Array Memory

As already described, we use a multi-array encoding for memory, with one array per block id, each indexed on $\widetilde{\mathsf{off}}$. A simpler encoding would have used a single array indexed on ptr. The multi-array encoding is beneficial when we can cheaply compute small aliasing sets for each memory access. In that case, we reduce the case-splitting work on bid that the SMT solver needs to do, and it enables further formula simplifications like store forwarding.

The multi-array encoding may, however, end up in a larger encoding overall if several of the accesses may alias with too many blocks. For load operations that alias multiple blocks the resulting expression is a linear combination of the loads of each block, e.g., $\mathsf{ite}(\mathsf{bid} = 0, \textbf{load}(m_0, \widetilde{\mathsf{off}}), \mathsf{ite}(\mathsf{bid} = 1, \textbf{load}(m_1, \widetilde{\mathsf{off}}), \ldots))$. In this case, it would be more compact to use the single-array encoding. Note that even if we do not know the specific block id, we often know whether a pointer refers to a local or non-local block (e.g., pointers received as argument have unknown block id, but are known to be non-local), and hence splitting the memory in two is usually a good idea (c.f. §10).

We perform several optimizations that are enabled with this multi-array encoding. We do partial-order reduction (POR) to shrink the potential aliasing of pointers with unknown block id. For example, consider a function with two pointer arguments (x and y) and one global variable. We assign $\mathsf{bid} = 1$ to the global variable. Then, we estipulate that x can only alias blocks with $\mathsf{bid} \leq 2$, which is sufficient to access the global variable or another unknown block. Argument y is also constrained to only alias blocks with $\mathsf{bid} \leq 3$, allowing it to alias with the global variable, the same block as x, or a different block. The same is done for function calls that return pointers. This POR technique greatly reduces the potential aliasing of unknown pointers without losing precision.

$\text{Num}(sz) ::= \{i \mid 0 \le i < 2^{sz}\}$	$\text{BlockID} ::= \mathbb{N}$	$\text{Addr} ::= \text{Num}(64)$	$\text{Offset} ::= \text{Num}(64)$
$\text{PtrAttr} ::= \{\texttt{nocapture}, \texttt{readonly}, \texttt{readnone}\}$		$\text{Pointer} ::= \text{BlockID} \times \text{Offset} \times 2^{\text{PtrAttr}}$	
$\text{Value} ::= \text{Aggregate} \uplus \text{Int} \uplus \text{Pointer} \uplus \text{Float} \uplus \{\textbf{poison}\}$		$\text{Aggregate} ::= \text{list Value}$	
$\text{PtrByte} ::= (\text{Pointer} \times \{i \mid 0 \le i < 8\}) \uplus \{\textbf{poison}\}$		$\text{NonPtrByte} ::= \text{Num}(8) \times \text{Num}(8)$	
$\text{Byte} ::= \text{PtrByte} \uplus \text{NonPtrByte}$	$\text{Bytes} ::= \text{Offset} \to \text{Byte}$	$\text{Size} ::= \text{Num}(64)$	
$\text{Align} ::= \{i \mid 0 \le i < 64\}$	$\text{Kind} ::= \{\texttt{stack}, \texttt{malloc}, \texttt{new}, \texttt{global}\}$	$\text{Live} ::= \text{bool}$	
$\text{Writable} ::= \text{bool}$	$\text{MemBlock} ::= \text{Addr} \times \text{Align} \times \text{Kind} \times \text{Live} \times \text{Writable} \times \text{Size} \times \text{Bytes}$		
$\text{Memory} ::= \text{BlockID} \to \text{MemBlock}$	$\text{UB} ::= \text{bool}$	$\text{FinalState} ::= \text{Value} \times \text{Memory} \times \text{UB}$	

$p \in \text{Pointer}$	$ag \in \text{Aggregate}$	$v \in \text{Value}$	$pb \in \text{PtrByte}$	$nb \in \text{NonPtrByte}$
$b \in \text{Byte}$	$mb \in \text{MemBlock}$	$M \in \text{Memory}$	$ub \in \text{UB}$	$\mu \in \text{BlockID} \nrightarrow \text{BlockID}$

Fig. 2. Type Definitions and Variable Naming Conventions.

$$\frac{v \in \text{Value}}{\textbf{poison} \sqsupseteq^{\mu} v}\ (\text{VALUE-POISON}) \qquad \frac{v \in \text{Int} \uplus \text{Float}}{v \sqsupseteq^{\mu} v}\ (\text{VALUE-NONPTR}) \qquad \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu} p'}\ (\text{VALUE-PTR})$$

$$\frac{|ag| = |ag'| \qquad \forall i,\ ag[i] \sqsupseteq^{\mu} ag'[i]}{ag \sqsupseteq^{\mu} ag'}\ (\text{VALUE-AGGREGATE})$$

$$\frac{}{(v, M, \textbf{true}) \sqsupseteq_{\text{st}} (v', M', ub')}\ (\text{FINAL-STATE-UB}) \qquad \frac{ub = ub' \qquad \exists \mu,\ v \sqsupseteq^{\mu} v' \wedge M \sqsupseteq^{\mu}_{\text{mem}} M'}{(v, M, ub) \sqsupseteq_{\text{st}} (v', M', ub')}\ (\text{FINAL-STATE})$$

Fig. 3. Refinement of value and final state.

7 Verifying Correctness of Optimizations

To verify correctness of LLVM optimizations, we establish a refinement relation between source (or original) and target (or optimized) functions. Equivalence is not used due to undefined behavior and nondeterminism. Compilers are allowed to reduce the set of possible behaviors from the source.

Given functions f_{src} and f_{tgt}, set of input and output variables I_{src}/I_{tgt} and O (which include, e.g., memory and the return value), and set of non-determinism variables N_{src}/N_{tgt}, f_{src} is refined by f_{tgt} iff:

$$\begin{aligned} \forall I_{src}, I_{tgt}, O_{tgt}.\ \ & \text{valid}(I_{src}, I_{tgt}) \ \wedge\ I_{src} \sqsupseteq I_{tgt} \ \wedge\ \exists N_{src}.\, \text{pre}_{\text{src}}(I_{src}, N_{src}) \wedge \\ & \big(\exists N_{tgt}.\, \text{pre}_{\text{tgt}}(I_{tgt}, N_{tgt}) \ \wedge\ [\![f_{tgt}]\!](I_{tgt}, N_{tgt}) = O_{tgt}\big) \\ \implies\ & (\exists N_{src}.\, \text{pre}_{\text{src}}(I_{src}, N_{src}) \ \wedge\ [\![f_{src}]\!](I_{src}, N_{src}) \sqsupseteq_{\text{st}} O_{tgt}) \end{aligned}$$

Predicate $\text{valid}(I_{src}, I_{tgt})$ encodes the global precondition of the input memory and arguments such as disjointness of non-local blocks. Function's preconditions, pre_{src} and pre_{tgt}, include the constraint for disjointness of local blocks. The existential pre_{src} constrains the input such that the source function has at least one possible execution. $\sqsupseteq_{\text{st}}$ is the refinement between final states.

Fig. 2 shows the definition of final program state which is a tuple of return value, return memory, and UB. A memory is a function from block id to a memory block. A memory block has seven attributes that are described in §4.3.

$$\frac{p.\text{block.live} \Rightarrow p'.\text{block.live} \quad p.\text{offset} = p'.\text{offset} \quad \left[\begin{array}{l}(\text{isNonLocal}(\{p,p'\}) \wedge p.\text{block.id} = p'.\text{block.id}) \\ \vee\ (\text{isLocal}(\{p,p'\}) \wedge p.\text{block.id} = \mu[p'.\text{block.id}])\end{array}\right]}{p \sqsupseteq^{\mu}_{\text{ptr}} p'}\ (\text{POINTER})$$

$$\frac{\left[\begin{array}{l}\forall bid,\ \text{isNonLocal}(bid) \\ \implies M[bid] \sqsupseteq^{\mu}_{\text{blk}} M'[bid]\end{array}\right] \quad \left[\begin{array}{l}\forall bid,\ \text{isLocal}(bid) \wedge \mu[bid]\ \text{defined} \\ \implies M[\mu[bid]] \sqsupseteq^{\mu}_{\text{blk}} M'[bid]\end{array}\right]}{M \sqsupseteq^{\mu}_{\text{mem}} M'}\ (\text{MEMORY-MAP})$$

$$\frac{pb.\text{byteoff} = pb'.\text{byteoff} \quad pb.\text{ptr} \sqsupseteq^{\mu}_{\text{ptr}} pb'.\text{ptr}}{pb \sqsupseteq^{\mu}_{\text{byte}} pb'}\ (\text{BYTE-PTR})$$

$$\frac{nb'.\text{p} \mid nb.\text{p} = nb.\text{p} \quad nb.\text{v} \mid nb.\text{p} = nb'.\text{v} \mid nb.\text{p}}{nb \sqsupseteq^{\mu}_{\text{byte}} nb'}\ (\text{BYTE-NONPTR})$$

$$\frac{\text{isZeroByte}(b) \quad \text{isZeroByte}(b')}{b \sqsupseteq^{\mu}_{\text{byte}} b'}\ (\text{BYTE-ZERO})$$

$$\frac{\text{isPoisonByte}(b)}{b \sqsupseteq^{\mu}_{\text{byte}} b'}\ (\text{BYTE-POISON})$$

$$\frac{\left[\begin{array}{l}\forall\, 0 \le i < mb.\text{size}, \\ mb.\text{bytes}[i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[i]\end{array}\right]}{mb \sqsupseteq^{\mu}_{\text{bytes}} mb'}\ (\text{BYTES})$$

$$\frac{\begin{array}{ll}mb.\text{live} \Rightarrow mb'.\text{live} & mb.\text{size} = mb'.\text{size} \\ mb.\text{kind} = mb'.\text{kind} & mb.\text{writable} = mb'.\text{writable} \\ mb.\text{align} \le mb'.\text{align} & mb.\text{live} \Rightarrow mb \sqsupseteq^{\mu}_{\text{bytes}} mb'\end{array}}{mb \sqsupseteq^{\mu}_{\text{blk}} mb'}\ (\text{BLOCK})$$

Fig. 4. Refinement of memory and pointers.

Fig. 3 shows the definition of refinement of value and final state. For pointers, we cannot simply use equality because local pointers in source and target are internal to each of the functions. Even if they have the same block identifier, they may refer to different allocation sites in the functions (VALUE-PTR). Similarly, the refinement of the final state should consider this difference between local pointers. To address this, we track a mapping μ between escaped local blocks of the two functions (described next).

7.1 Refinement of Memory

Checking refinement of non-local memory blocks is simple as blocks are the same in the source and target functions (e.g., global variables have the same ids in the two functions). Therefore, one just needs to compare blocks of source and target functions with the same id pairwise.

Checking refinement of local blocks is harder but needed when, e.g., the function returns a locally-allocated heap block. This is legal, but block ids in the two functions may not be equal as allocations may have happened in a different order. Therefore, we cannot simply compare local blocks with the same ids.

To check refinement of local blocks, we need to align the two functions' allocations, i.e., we need to find a correspondence between local blocks of the two functions. We introduce a mapping $\mu \in \text{BlockID} \nrightarrow \text{BlockID}$ between target and source local block ids.

Local blocks become related on function calls and return statements, which is when local pointers may be observed. For example, if a function is called with a pointer to a local block as the first argument, μ should relate that pointer with the first argument of an equivalent function call in the target function.

$$\frac{\left[\begin{array}{l} v, v' \notin \\ \text{Pointer} \end{array}\right] \quad v \sqsupseteq^{\mu} v'}{v \sqsupseteq^{\mu,sz}_{\text{arg}} v'}\ (\text{NONPTR-ARG}) \qquad \frac{p \sqsupseteq^{\mu}_{\text{ptr}} p'}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}\ (\text{PTR-ARG-MAPPED}) \qquad \frac{\begin{array}{c}\text{isLocal}(\{p, p'\}) \\ p.\text{offset} = p'.\text{offset} \\ M[p.\text{bid}] \sqsupseteq^{\mu}_{\text{blk}} M'[p'.\text{bid}]\end{array}}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}\ (\text{PTR-ARG-UNMAPPED})$$

$$\frac{\begin{array}{c} sz > 0 \quad o = p.\text{offset} \quad o' = p'.\text{offset} \\ mb = M[p.\text{bid}] \quad mb' = M'[p'.\text{bid}] \\ \left[\begin{array}{l} \forall 0 \le i < sz, \\ mb.\text{bytes}[o+i] \sqsupseteq^{\mu}_{\text{byte}} mb'.\text{bytes}[o'+i] \end{array}\right] \end{array}}{p \sqsupseteq^{\mu,sz}_{\text{arg}} p'}\ (\text{PTR-ARG-BYVAL})$$

Fig. 5. Refinement between function arguments.

Fig. 4 gives the definition of memory refinement, $M \sqsupseteq^{\mu}_{\text{mem}} M'$, as well as other related relations between memory blocks and pointers. The first rule POINTER describes refinement between source pointer p and target pointer p' with respect to μ. The following four rules define refinement between bytes b and b'. In rule BYTE-NONPTR, '$a \,|\, b$' is the bitwise OR operation, and it is used to check the equality of only those bits that are not **poison**. Predicate isZeroByte(b) holds if b is a **null** pointer or if it is a zero-valued non-pointer byte. This is needed because stores of **null** pointers can be optimized to **memset** instructions.

Rules BYTES and BLOCK define refinement between memory blocks' values and memory blocks, respectively. Rule MEMORY-MAP describes memory refinement with respect to local block mapping μ. $M[bid]$ stands for the memory block with block id *bid*.

The well-formedness of μ is established in the refinement rules for function calls and return statements. We show these for function calls in the next section. We note that there might be multiple well-formed μ due to non-determinism.

8 Function Calls

A call to an unknown function may change the memory arbitrarily (except for, e.g., constant variables and non-escaped local blocks). The outputs in the source and target are, however, related: if the target's inputs refine those of the source, refinement holds between their outputs as well. Alive2 already supported function calls; this section shows how it was extended to support memory.

Let (M_{in}, v_{in}) and (M_{out}, v_{out}) be the input and output of a function call in the source, and their primed versions, (M'_{in}, v'_{in}) and (M'_{out}, v'_{out}), those of a function call in the target. Let μ_{in} be a local block mapping before executing the calls. To state that the outputs are refined if the inputs are refined, we add the following formula to the target's precondition:

$$\Big(M_{in} \sqsupseteq^{\mu_{in}}_{\text{mem}} M'_{in} \wedge \forall i\,.\; v_{in}[i] \sqsupseteq^{\mu_{in}, sz[i]}_{\text{arg}} v'_{in}[i]\Big) \Longrightarrow \big(M_{out} \sqsupseteq^{\mu_{out}}_{\text{mem}} M'_{out} \wedge v_{out} \sqsupseteq^{\mu_{out}} v'_{out}\big)$$

A call to a function with a pointer to a local block as argument escapes this block, as the callee may, e.g., store that pointer to a global variable. Moreover, any pointer stored in this block also escapes as the callee may traverse the block and grab any pointer stored there, and do so transitively. The updated mapping $\mu_{out} = \text{extend}(\mu_{in}, M_{in}, M'_{in}, v_{in}, v'_{in})$ returns μ_{in} updated with the relationship between the newly escaped blocks in source and target functions.

Fig. 5 shows the definition of refinement between function call arguments in source and target programs. The first rule relates non-pointer arguments. The second one handles pointers that have escaped before these calls. The third rule handles local pointers of blocks that did not escape before these calls, and therefore we need to check if the contents of these block are refined.

The fourth refinement rule handles **byval** pointer arguments. These arguments get a freshly allocated block and the contents of the pointer are copied from the pointer's offset onwards.

9 Approximating Program Behavior

In order to speedup verification, we approximate programs' behaviors, which can result in false positives and false negatives. We believe none of these approximations has a significant impact for two reasons: (1) we only need to be as precise as LLVM's static analyses, i.e., we do not need to support arbitrary optimizations, and (2) we do not consider the compiler to be malicious (which may not be true in certain contexts). Moreover, we conducted an extensive evaluation to support these claims, on which we report in the next section.

Under-Approximations

1. Physical addresses of local memory blocks have the MSB set to 1, and non-locals set to 0. This is reasonable if we assume the compiler is not malicious and therefore will not exploit our approximation.
2. We do not consider the case where a (portion of a) global variable is initially **undef**, only **poison** or a regular value.
3. Library functions **strlen**, **memcmp**, and **bcmp** are unrolled for a constant number of times. A precondition is added to constrain the input to be smaller than the unroll factor. In the case of **strlen**, the input pointer is often a constant array. We compute the result straight away in this case.

Over-Approximations. The set of local blocks that escape (e.g., whose address is stored into a global variable) is computed per function. This may over-approximate the set of escaped pointers at times because, e.g., a pointer may only escape in a particular branch. LLVM also computes the set of escaped pointers per function.

10 Evaluation

We implemented our new memory model in Alive2 [30]. The implementation of the memory model consists in about 3.0 KLoC plus an additional 0.4 KLoC for static analyses for optimization.

We run two set of experiments to both validate our implementation and the formal semantics, and to identify bugs in LLVM. First, we did translation validation of LLVM's unit tests (`test/Transforms`) to increase confidence that

Program	LoC	Pairs	Time (hours)	Correct	Incorrect	TO	OOM	Unsupported pairs
bzip2	5.1k	2.3k	1.9	316	9	574	175	1.2k
gzip	5.3k	2.6k	2.0	908	4	922	45	737
oggenc	48k	1.8k	2.0	433	5	617	49	701
ph7	43k	5.6k	3.4	1.2K	23	1.5K	15	2.8k
sqlite3	141k	12k	7.5	2.2k	38	2.2K	48	7.8k

Table 3. Statistics and results for the single-file benchmarks.

we match LLVM's behavior in practice. Second, we run five benchmarks: bzip2, gzip, oggenc, ph7, and SQLite3.

Benchmarks were compiled with `-O3`. Moreover, we disabled type-based aliasing because there is no formal model for this feature yet. During compilation, we emitted pairs of IR files before and after each intra-procedural optimization. We discarded syntactically equal pairs as well as pairs without memory operations.

We used a machine with two Intel Xeon E5-2630 v2 CPUs (total of 12 cores). We set Z3's timeout to 1 min and memory limit to 1 GB. Loops were unrolled once. We used LLVM from 11/Dec (`5e31e22`) and Z3 [33] from 16/Dec (`11477f`).

10.1 LLVM Unit Tests

LLVM's `Transforms` unit test suite consists in 6,600 tests totaling 36,600 functions. Alive2 takes about 2.5 hours (in parallel) to validate these. By running LLVM's unit tests, we found 21 new bugs in memory optimizations.

We show below an example of a bug we found. This optimization was shrinking the store from 64 to 32 bits, which is incorrect since the last 32 bits were not copied. This happened because of the mismatch in the load/store's sizes.

```
// i32 *x, *y, *z;                  // i32 *x, *y, *z;
i32 *p = (*x < *y ? x : y);   ⇏     i32 r = (*x < *y ? *x : *y);
*(i64*)z = *(i64*)p;                *z = r;
```

10.2 Benchmarks

Table 3 shows the statistics and results for translation validation. The Pairs column indicates the number of source/optimized function pairs considered for validation. We discarded pairs where the two functions were syntactically equal, as the transformation is then trivially correct. The last column indicates the number of skipped pairs because they use features Alive2 does not yet support.

All the 79 incorrect pairs are due to mismatches between LLVM and the formal semantics. Of these, 74 are related with incorrect handling of **undef** and **poison** values, and the remaining 5 are caused by incorrect load type punning optimizations. This shows that our tool has no false positives.

10.3 Specification Bugs

While testing our tool, we found a mismatch in the semantics of the **nonnull** attribute between LLVM's documentation and LLVM's code. The documentation

specified that passing a null pointer to a **nonnull** argument triggered UB. However, as illustrated below, LLVM adds **nonnull** to a pointer that may be **poison**. This is incorrect because **poison** can be optimized into any value including null.

```
p = gep inbounds q, 1          ⇒     p = gep inbounds q, 1
f(p)                                 f(nonnull p) ; UB if p poison
```

We proposed a new semantics to the LLVM developers, where non-conforming pointers would be considered **poison** rather than UB. This was accepted and we have contributed patches to fix the docs and the incorrect optimizations.

10.4 Alias Sets

To show that splitting the memory into multiple arrays is beneficial, we gathered statistics of the alias sets in our benchmarks. More than 96% of the dereferenced pointers turned out to be only local or non-local, but not both. This shows that splitting the memory into local and non-local simplifies the memory encoding.

We also counted the number of memory blocks pointers may alias with. Half of the pointers were aliased with just one block. About 80% of the pointers aliased with at most 3 blocks. This is much less than the median number of blocks functions have. The median of the number of memory blocks was $7 \sim 13$ (varying over programs), and only 10% of the functions had fewer than 3 blocks.

11 Related Work

Semantics of LLVM IR. The official LLVM IR's specification is written in prose [1]. Vellvm [47] and K-LLVM [29] formalized large subsets of the IR in Coq and K, respectively. [26] clarifies the semantics of **undef** and **poison** and proposes a new **freeze** instruction. [24] formalizes various memory instructions of LLVM. [32] presents a C memory model that supports compilation to that LLVM model.

Translation validation. [38] presents a translation validation infrastructure for GCC's intermediate language, using a set of arithmetic/aliasing rules for showing equivalence. LLVM-MD [44] and Peggy [42] verify LLVM optimizations by showing equivalence of source and targets with rewrite rules/equality axioms. They suffer, however, from incomplete axioms for aliasing.

In order to simplify the work of translation validation tools, it is possible to extend the compiler to produce hints (witnesses) [18, 36, 38, 41]. One of these tools, Crellvm [20], is formally verified in Coq.

Verifying programs with memory using SMT solvers. SMT solvers have been used before to check equivalence of programs with memory [11, 14, 21, 25, 31]. [12] give an encoding of some (but not all) aliasing constraints needed to do translation validation of assembly generated by C compilers.

Other memory models encoded in SMT include one for Solidity (Etherium smart contracts) [16], and for separation logic [37, 39]. Several verification tools include SAT/SMT-based (partial) memory models for C [2, 9, 10] and Java [43].

Several automatic software verification tools, often based on CHCs (constrained Horn clauses), support memory programs [6,13]. For example, both SeaHorn and Cascade use a field-sensitive alias analysis to split the memory [15,45]. SLAyer [4] is an automatic tool for analyzing memory safety of a C program using Z3. Smallfoot [3] verifies assertions written in separation logic.

There have been recent advances in speeding up verification of (SMT) array programs [17, 22], from which we could likely benefit.

CompCert [27] splits the memory into local (private) and non-local (public) blocks, similarly to what we do, but assumes that allocations never fail [28]. Work on verifying peephole optimizations for CompCert does not support memory [34].

To support integer-to-pointer casts in CompCert, [5] proposes extending integer values to carry block ids as well. In this model, arithmetic on pointer values yields a symbolic expression. [19] makes the pointer-to-integer cast an instruction that assigns a physical address to the block. Neither of these models supports several optimizations performed by LLVM.

12 Conclusion

We presented the first SMT encoding of LLVM's memory model that is sufficiently precise to validate all of LLVM's intra-procedural memory optimizations.

Using our new encoding, we found and reported 21 previously unknown bugs in LLVM memory optimizations, 10 of which have already been fixed.

Acknowledgements This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF-2020R1A2C2011947).

References

1. LLVM language reference manual, `https://llvm.org/docs/LangRef.html`
2. Ball, T., Bounimova, E., Levin, V., de Moura, L.: Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Tech. Rep. MSR-TR-2010-24, Microsoft Research (2010), `https://www.microsoft.com/en-us/research/publication/efficient-evaluation-of-pointer-predicates-with-z3-smt-solver-in-slam2/`
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2006). https://doi.org/10.1007/11804192_6
4. Berdine, J., Cook, B., Ishtiaq, S.: Slayer: Memory safety for systems-level code. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-1_15
5. Besson, F., Blazy, S., Wilke, P.: A concrete memory model for CompCert. In: ITP (2015). https://doi.org/10.1007/978-3-319-22102-1_5

6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_8
7. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In: CAV (2002). https://doi.org/10.1007/3-540-45657-0_7
8. Chakraborty, S., Vafeiadis, V.: Formalizing the concurrency semantics of an LLVM fragment. In: CGO (2017). https://doi.org/10.1109/CGO.2017.7863732
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE (2009). https://doi.org/10.1109/ASE.2009.63
11. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: APLAS (2017). https://doi.org/10.1007/978-3-319-71237-6_7
12. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: HVC (2017). https://doi.org/10.1007/978-3-319-70389-3_2
13. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012). https://doi.org/10.1145/2254064.2254112
14. Gupta, S., Saxena, A., Mahajan, A., Bansal, S.: Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In: SAT (2018). https://doi.org/10.1007/978-3-319-94144-8_22
15. Gurfinkel, A., Navas, J.A.: A context-sensitive memory model for verification of C/C++ programs. In: SAS (2017). https://doi.org/10.1007/978-3-319-66706-5_8
16. Hajdu, Á., Jovanović, D.: SMT-friendly formalization of the solidity memory model. In: ESOP (2020)
17. Ish-Shalom, O., Itzhaky, S., Rinetzky, N., Shoham, S.: Putting the squeeze on array programs: Loop verification via inductive rank reduction. In: VMCAI (2020). https://doi.org/10.1007/978-3-030-39322-9_6
18. Kanade, A., Sanyal, A., Khedker, U.P.: Validation of GCC optimizers through trace generation. SP&E **39**(6), 611–639 (Apr 2009). https://doi.org/10.1002/spe.913
19. Kang, J., Hur, C.K., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: PLDI (2015). https://doi.org/10.1145/2737924.2738005
20. Kang, J., Kim, Y., Song, Y., Lee, J., Park, S., Shin, M.D., Kim, Y., Cho, S., Choi, J., Hur, C.K., Yi, K.: Crellvm: Verified credible compilation for LLVM. In: PLDI (2018). https://doi.org/10.1145/3192366.3192377
21. Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification of pointer programs by predicate abstraction. Formal Methods in System Design **52**(3), 229–259 (Jun 2018). https://doi.org/10.1007/s10703-017-0293-8
22. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional verification of procedural programs using horn clauses over integers and arrays. In: FMCAD (2015). https://doi.org/10.1109/FMCAD.2015.7542257
23. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. In: PLDI (2014). https://doi.org/10.1145/2594291.2594334
24. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code in LLVM. Proc. of the ACM on Programming Languages **2**(OOPSLA) (Nov 2018). https://doi.org/10.1145/3276495
25. Lee, J., Hur, C.K., Lopes, N.P.: AliveInLean: A verified LLVM peephole optimization verifier. In: CAV (2019). https://doi.org/10.1007/978-3-030-25543-5_25

26. Lee, J., Kim, Y., Song, Y., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming undefined behavior in LLVM. In: PLDI (2017). https://doi.org/10.1145/3062341.3062343
27. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814
28. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert memory model, version 2. Tech. Rep. RR-7987, INRIA (Jun 2012), `http://hal.inria.fr/hal-00703441`
29. Li, L., Gunter, E.L.: K-LLVM: A relatively complete semantics of LLVM IR. In: ECOOP (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.7
30. Lopes, N.P., Lee, J., Hur, C.K., Liu, Z., Regehr, J.: Alive2: Bounded translation validation for LLVM. In: PLDI (2021). https://doi.org/10.1145/3453483.3454030
31. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: PLDI (2015). https://doi.org/10.1145/2737924.2737965
32. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C semantics and pointer provenance. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290380
33. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008). https://doi.org/10.1007/978-3-540-78800-3_24
34. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: PLDI (2016). https://doi.org/10.1145/2908080.2908109
35. Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: A proof of concept. In: RV (2013). https://doi.org/10.1007/978-3-642-40787-1_22
36. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_17
37. Navarro Pérez, J.A., Rybalchenko, A.: Separation logic modulo theories. In: APLAS (2013). https://doi.org/10.1007/978-3-319-03542-0_7
38. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000). https://doi.org/10.1145/349299.349314
39. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: CAV (2013). https://doi.org/10.1007/978-3-642-39799-8_54
40. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS (1998). https://doi.org/10.1007/BFb0054170
41. Rinard, M.C., Marinov, D.: Credible compilation with pointers. In: RTRV (1999)
42. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: CAV (2011). https://doi.org/10.1007/978-3-642-22110-159
43. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking axiomatic specifications of memory models. In: PLDI (2010). https://doi.org/10.1145/1806596.1806635
44. Tristan, J.B., Govereau, P., Morrisett, J.G.: Evaluating value-graph translation validation for LLVM. In: PLDI (2011). https://doi.org/10.1145/1993316.1993533
45. Wang, W., Barrett, C., Wies, T.: Partitioned memory models for program analysis. In: VMCAI (2017). https://doi.org/10.1007/978-3-319-52234-0_29
46. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI (2011). https://doi.org/10.1145/1993498.1993532
47. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL (2012). https://doi.org/10.1145/2103656.2103709
48. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formal verification of SSA-based optimizations for LLVM. In: PLDI (2013). https://doi.org/10.1145/2491956.2462164