A Precise Memory Model for Low-Level Bounded Model Checking

Carsten Sinz Stephan Falke Florian Merz
Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{carsten.sinz, stephan.falke, florian.merz}@kit.edu

***Abstract*—Formalizing the semantics of programming languages like C or C++ for bounded model checking can be cumbersome if complete coverage of all language features is to be achieved. On the other hand, low-level languages that occur during translation (compilation) have a much simpler semantics since they are closer to the machine level. It thus makes sense to use these low-level languages for bounded model checking. In this paper we present a highly precise memory model suitable for bounded model checking of such low-level languages. Our method also takes memory protection (*malloc*/*free*) into account.**

I. Introduction

Bounded model checking (BMC) has proven to be a very successful technique in hardware verification [10]. More recently, it has also been applied for verifying software written in C [6], [14]. Applying BMC for verifying C programs, however, comes with many obstacles that have to be tackled. One of the most important differences is that the syntax (and thus semantics) of a programming language like C is much more complicated than a hardware description. One has to deal, e.g., with memory allocation, (function) pointers, complex data structures, and function calls. Going from C to C++ complicates things even further. To analyze a line of C++ source code like

```
complex i(0.0,-1.0), a = 2.0 + i;
```

may involve constructors, overloaded operators, and user-defined conversion functions. Thus, the precise semantics of such a statement can only be determined by taking further parts of the source code into account.

We therefore propose a new approach which, instead of exploring the source code directly, makes use of existing compiler technology and performs the analysis on an assembler-like compiler intermediate language instead. Such an intermediate language offers a much simpler syntax (and semantics), and thus eases a logical encoding of the verification problem considerably.

This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

We have chosen the LLVM (Low Level Virtual Machine) [15] compiler infrastructure and its abstract assembler intermediate language as the starting point for our approach, but the idea can also be applied to other low-level languages. LLVM is both a (GCC-compatible) C/C++/Objective-C compiler and a library of compiler technologies, providing, e.g., source and target-independent optimizations.

The intermediate language of the LLVM framework is a typed, assembler-like language for a register machine (mainly three-address-code instructions working on an unbounded number of registers), offering static single assignment (SSA) form for scalar values (see Fig. 6 for an example of LLVM's intermediate language). LLVM's optimization features and its machine model (unlimited number of registers, SSA form) make it especially suitable for program analysis using BMC.

Our primary goal is to detect memory errors in C code. Memory errors include invalid memory accesses (reads or writes to memory locations that have not been allocated before), heap and stack buffer overflows, memory leaks, and invalid frees (e.g., double frees).

The overall structure of our BMC approach is depicted in Fig. 1. It makes use of the LLVM compiler front end and an SMT solver that handles the generated decision problems. Encoding program checks requires a transformation from the abstract assembler intermediate language into the logic of bit-vectors and arrays. Furthermore, a memory model which captures the semantics of memory accesses in C is needed at this point. The present paper mainly deals with how such a memory model can be constructed.

II. Basic Notions

A. Software Bounded Model Checking

Let us briefly recapitulate the main ideas of bounded model checking of software. Software inherently deals with unbounded data structures such as linked lists or trees, and may give rise to infinite program runs (e.g. in reactive or interactive systems). In general checking properties of such programs is undecidable. Bounded model checking thus limits all program runs

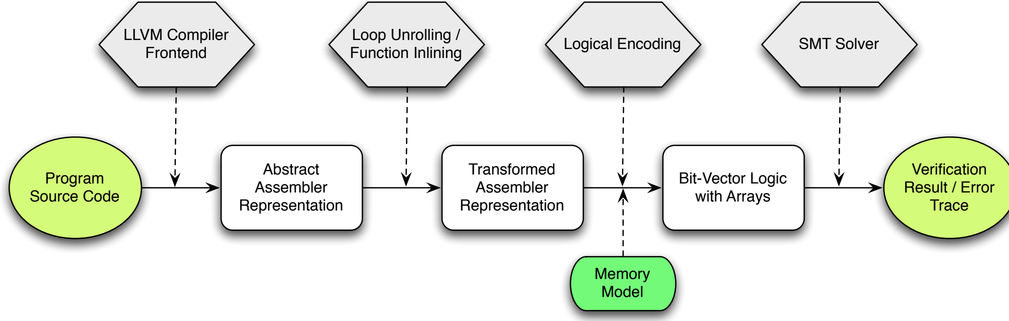

Fig. 1. Overview of our approach for low-level bounded model checking of C programs.

and data structures to finite ones, thereby achieving decidability. This is accomplished by analyzing only bounded program runs. The bound is typically imposed by restricting the number of nested function calls and loop iterations that are allowed. By considering only finite program runs, affected data structures also become finite. Function inlining and loop unrolling then results in one (rather large) function that is subject to verification.

In BMC, properties of a program are typically formalized using `assume`/`assert` statements, where an `assume` statement, inserted at some location in the program, formulates a pre-condition that is assumed to hold at this location: all program runs not satisfying this condition are cut off at the `assume` statement, hence the term bounded model checking. Similarly, `assert` statements encode post-conditions, i.e. properties that have to hold at the respective location. Assertions that do not hold constitute errors in the program.

B. LLVM Intermediate Representation

A program in LLVM abstract assembler consists of type definitions, global variable declarations, and the program itself, represented as a graph of basic blocks (see Fig. 6 for an example of an LLVM program). Each basic block is a list of instructions; (possibly conditional) jumps are only allowed as the last instruction of each block. The jump (or branch) instructions between basic blocks induce the *basic block graph* (Fig. 2 depicts it for the LLVM program in Fig. 6). We annotate edges in the basic block graph with the condition under which the transition between the two blocks is taken.[1]

For our purpose, LLVM instructions can be categorized into six classes:

- *Three-address-code (TAC)* instructions working on registers or constants, like $r_2 =$ `add i32` $r_1, 5$.
- *Memory access and allocation* instructions, namely `load`, `store`, `malloc`, and `free`.

[1]If an edge possesses no label, the transition is always taken.

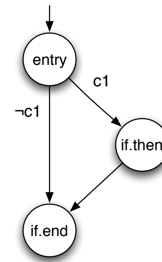

Fig. 2. Basic block graph for the program in Fig. 6.

- *Address calculations* using `getelementptr`.
- *Branch and* `phi` instructions.
- *Function calls.*
- *Auxiliary instructions* like type casts (type casts do not change the bit-level representation of the data).

Programs in LLVM assembler are in *static single assignment (SSA)* form, i.e. each (scalar) variable is assigned exactly once. Assignments to scalar variables can thus be treated as logical equivalences.

Variables in LLVM assembler are typed. Available types include integer types (like `i32`; the bit-width is given explicitly, but no distinction between signed and unsigned integers is made), floating-point types, and derived types (encompassing pointer, function, array, structure and union types). Aggregate types (structures, arrays, and unions) are accessed using memory read/write operations and offset calculations using the `getelementptr` instruction. Thus having a memory model for read and write operations and a translation for the `getelementptr` instruction is sufficient for handling all aggregate types.

Function inlining and loop unrolling can be accomplished using code provided by the LLVM libraries. Notice that the basic block graph becomes a DAG after loop unrolling.

To report error traces back to the user, we employ debug information generated by LLVM.

C. Logical Encoding

We now describe how an LLVM program can be transformed into a formula in the logic of bit-vectors and arrays. The most complex part of this transformation is the translation of memory instructions, which will be described in detail in the next section. How the remaining instructions are handled is explained in the following.

Translation of three-address-code LLVM instructions is straightforward since these instructions (like, e.g., `add`, `cmp`, `mul`, sign or zero extension) are supported by SMT solvers and assignments can be regarded as equalities due to the use of SSA form.

1) Handling of `phi` *Instructions:* `phi` instructions are a common tool in compiler technology when using static single assignment form. They are used to select the correct value for a variable out of a set of previous ones (e.g., when control merges again after an `if-then-else` statement; see Fig. 6 for an example). In general, a `phi` instruction has the form

$$i' = \texttt{phi}\ [i_1, c_1] \dots [i_n, c_n]$$

where the value of variable i' takes one of the values of $i_1, \dots, i_n$ depending on which of the conditions $c_1, \dots, c_n$ is true. The conditions c_j are mutually exclusive and cover all possible cases such that the value of i' is always uniquely determined.

A `phi` instruction can be translated into a sequence of ITE (if-then-else) operators (written in C syntax), which are also supported by SMT solvers:

$$i' = c_1\ ?\ i_1 : (c_2\ ?\ i_2 : (\dots(c_{n-1}\ ?\ i_{n-1} : i_n)\dots))$$

What complicates matters slightly when using LLVM is that in the assembler intermediate representation the conditions c_j are not given explicitly in the `phi` instructions, but basic blocks are used as designators instead. The basic blocks refer to the predecessor in the basic block graph from which the current basic block has been entered (see Fig. 6 for an example). We thus have to compute the conditions c_j for each basic block of a `phi` instruction. This is accomplished as follows. With each basic block b we associate an execution condition $c_{\texttt{exec}}(b)$. The execution condition can be calculated recursively. Let us denote by $P(b)$ the set of predecessors of b in the basic block graph, and by $t(b, b')$ the condition under which the transition from basic block b to b' is taken (the edge label of the basic block graph). Then

$$c_{\texttt{exec}}(b) = \bigvee_{\hat{b} \in P(b)} \left(c_{\texttt{exec}}(\hat{b}) \wedge t(\hat{b}, b)\right)$$

if $P(b) \neq \emptyset$, and $c_{\texttt{exec}}(b) = \texttt{true}$ otherwise. We can then replace the basic block $\hat{b}$ in a `phi` instruction occurring in basic block b by the condition $c_{\texttt{exec}}(\hat{b}) \wedge t(\hat{b}, b)$.

Notice that each definition of $c_{\texttt{exec}}(b)$ requires only linear space in the number of predecessors of the basic block b since we do not unroll the recursive definition but introduce a new Boolean (one-bit) variable for each $c_{\texttt{exec}}(b)$ and $t(b, b')$ instead.

2) The `getelementptr` *Instruction:* This instruction has the form $q = \texttt{getelementptr}\ p, o_1, \dots, o_k$, where p, q are pointers, and the o_i's are offsets. The offsets are either array indices or select the indicated element of a structure. In our translation, the `getelementptr` instruction is simply transformed into a linear equation $q = p + o_1 \cdot S_1 + \dots + o_k \cdot S_k$, where the constant multiplicands S_i can be computed easily based on the data type definitions and the type of pointer p.

3) Elimination of Branches: With the introduction of `phi` instructions and SSA form, branches are no longer needed. This is, however, only true if all instructions are in SSA form (and are thus state-independent). Since memory accesses (`load`/`store`) are still state-dependent, we have to remove these dependencies in order to eliminate branches completely. How this is accomplished is discussed in the next section.

III. A C-like Memory Model for LLVM

The aim of the memory model is to capture the C semantics as precisely as possible.[2] We thus do not employ a typed memory model (as, e.g., suggested by Cohen *et al.* [7]), but an untyped one, where arbitrary data can be read from any valid address. Such an untyped memory model supports constructs which commonly occur in low-level C code, e.g. casting a block of memory containing a structure to a byte-array for writing it to disk or sending it over the network.

In our approach memory is thus just an array of bytes. Stores and loads (of data with arbitrary size) are accomplished by a sequence of reads and writes to the memory array on the logic level. E.g., reading a 16 bit integer (x, say) from address p is encoded as

$$x = \texttt{read16}(m, p)$$

which is then converted into

$$x_{0..7} = \texttt{read}(m, p) \wedge x_{8..15} = \texttt{read}(m, p+1)$$

where m is the memory array, and $x_{i..j}$ stands for bits i to j of bit-vector x. At this point, a decision has to be made whether a little-endian or big-endian architecture is analyzed (we have assumed little-endian in our example).

The logical encoding uses the theory of arrays and thereby makes the state dependency of loads and stores

[2] In this paper, we only consider memory consistency checks for heap-allocated memory. Checks for global variables and stack-allocated memory can be handled similarly, however.

explicit. An instruction like `store i32 5, p` is converted into an array update with explicit mentioning of the modified (memory) array, resulting in the equation $m' = \texttt{write}(m, p, 5)$, where m' is the modified array after the write operation.

The memory model also has to keep track of allocated memory blocks, as access to un-allocated addresses is considered invalid. We therefore introduce a memory state type called `mem` which adds this information to the byte array.

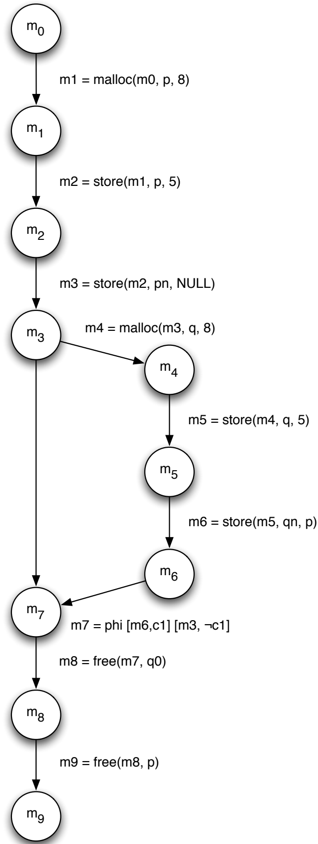

Fig. 3. Memory modification graph for the program in Fig. 6.

The type `mem` is abstract in the sense that we do not maintain an explicit representation of allocated memory blocks. Instead, we keep track of allocated and freed memory regions by means of the history or evolution of the memory state. We therefore introduce a *memory modification graph*, which consists of memory states as nodes, and two memory states are connected if a memory modification (`write`, `malloc`, or `free`) on the source state results in the target state (see Fig. 3 for an example). We also introduce `phi` nodes for memory states in this graph, which serve the same purpose as ordinary `phi` instructions for scalar values.

Fig. 7 (left) shows the encoding of our example program (Fig. 6) with explicit memory state transformations.[3]

A. Encoding Memory Constraints

Based on the memory modification graph we are now able to construct memory consistency constraints. Our goal is to encode memory consistency checks of the following kind:

1) Valid reads/writes (i.e. they affect only allocated memory).
2) Valid frees (i.e. the pointer given as a parameter to a `free` instruction points to the start of a currently allocated memory block).
3) No double frees (i.e. no memory is de-allocated twice).
4) No memory leaks (i.e. all allocated heap memory is de-allocated when the program ends).

We use `malloc` and `free` instructions with signatures

$$m' = \texttt{malloc}(m, p, s) \quad \text{and}$$
$$m' = \texttt{free}(m, p)$$

where m, m' are memory states, p is a pointer, and s is the size (in bytes) of the memory block to be allocated. The signature of the `malloc` instruction seems a bit unusual, as it takes the pointer p as a parameter. We use the `malloc` instruction in such a way that it is always preceded by a new pointer variable declaration for p, and `malloc` adds suitable constraints to this pointer. The memory state m' returned by `malloc` can then be considered as having these constraints added. Notice that `malloc` instructions may also fail (if sufficient memory is not available); the pointer p is then set to `NULL`. The `free` instruction modifies the memory state in such a way that the (currently allocated) memory block starting at address p is deallocated. A `NULL` pointer may also be passed to `free`, the instruction then has no effect.

To formulate the memory constraints we introduce further notation. For two memory states, m and m', by $m \preceq m'$ we denote that there is a path from m to m' in the memory modification graph. For a memory-modifying instruction I we write $c_{\texttt{exec}}(I)$ for the execution condition of the basic block that I belongs to. With these notions, we can define predicates on the memory state (the formulas for these predicates are given in Fig. 4):

- $\texttt{valid_mem_access}(m, p, s)$ denotes whether access to the memory range $p, \ldots, p + s - 1$ in memory state m is admissible.

[3]Basic block annotations such as their execution conditions are still needed in order to formulate memory constraints, as we will see later.

$$\texttt{valid_mem_access}(m,p,s) \equiv \bigvee_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\hat{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge q \neq \texttt{NULL} \wedge (q \leq p \leq q+t-s) \wedge \neg\texttt{deallocated}(m',m,q) \Big)$$

$$\texttt{deallocated}(m,m',p) \equiv \bigvee_{\substack{m \preceq m^* \preceq m' \\ I:\, m^* = \texttt{free}(\hat{m}^*,q)}} \Big(c_{\texttt{exec}}(I) \wedge p = q \Big)$$

$$\texttt{non_overlap}(p,s,q,t) \equiv (p+s \leq q) \vee (q+t \leq p)$$

$$\texttt{malloc_assumption}(m,p,s) \equiv p = \texttt{NULL} \vee \Big[(p \geq \texttt{MEM_MIN} \wedge p+s-1 \leq \texttt{MEM_MAX}) \wedge \bigwedge_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\hat{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge q \neq \texttt{NULL} \wedge \neg\texttt{deallocated}(m',m,q) \implies \texttt{non_overlap}(p,s,q,t) \Big) \Big]$$

$$\texttt{valid_free}(m,p) \equiv p = \texttt{NULL} \vee \bigvee_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\hat{m},q,t)}} \Big(c_{\texttt{exec}}(I) \wedge p = q \wedge \neg\texttt{deallocated}(m',m,q) \Big)$$

$$\texttt{no_memory_leaks}(m) \equiv \bigwedge_{\substack{m' \preceq m \\ I:\, m' = \texttt{malloc}(\hat{m},p,s)}} \Big[c_{\texttt{exec}}(I) \implies \bigvee_{\substack{m' \preceq m^* \preceq m \\ J:\, m^* = \texttt{free}(\hat{m}^*,q)}} \Big(c_{\texttt{exec}}(J) \wedge p = q \Big) \Big]$$

Fig. 4. Memory consistency predicates.

- $\texttt{deallocated}(m, m', p)$ denotes whether the memory block starting at address p is deallocated (`freed`) between memory states m and m'.
- $\texttt{non_overlap}(p, s, q, t)$ denotes that the memory ranges $p, \ldots, p+s-1$ and $q, \ldots, q+t-1$ do not overlap.
- $\texttt{malloc_assumption}(m, p, s)$ denotes the assumptions that can be made on pointer p after a `malloc` instruction that produced memory state m has been executed.
- $\texttt{valid_free}(m, p)$ denotes that a `free` instruction on pointer p in memory state m is admissible. This includes two aspects: first, the memory region must have been allocated before and, second, it must not have been deallocated prior to this `free` instruction (i.e., not double free occurs).
- $\texttt{no_memory_leaks}(m)$ expresses that in memory state m all heap-allocated memory has been deallocated again.

Having these definitions, memory checks can be formalized as follows.

1) Valid Reads and Writes: For this check, we first add an $\texttt{assume}(\texttt{malloc_assumption}(m', p, s))$ statement after each $m' = \texttt{malloc}(m, p, s)$ instruction. Then we add $\texttt{assert}(\texttt{valid_mem_access}(m, p, s))$ before each $m' = \texttt{write}(m, p, x)$ and $x = \texttt{read}(m, p)$ instruction, where s is the size in bytes of the data to be read or written. Having processed all `malloc`s, `read`s and `write`s, the `malloc` and `free` instructions can be removed by replacing $m' = \texttt{malloc}(m, p, s)$ by $m' = m$, and similarly for `free`s.

In order to illustrate our formalization, we compute the `malloc_assumption` for the second `malloc` instruction in our example program (see Fig. 7):

$$\begin{aligned} &\texttt{malloc_assumption}(m_4, q, 8) \\ \equiv\ & q = \texttt{NULL} \vee (q \geq \texttt{MEM_MIN} \wedge q+7 \leq \texttt{MEM_MAX}) \wedge \\ & (p \neq \texttt{NULL} \implies (q+8 \leq p \vee p+8 \leq q)) \end{aligned}$$

Here, by `MEM_MIN` and `MEM_MAX` we denote the minimal and maximal addresses of valid heap space. By dropping the tests for pointer-equality with `NULL`, unfailing `malloc` instructions can be modelled.

As a further example, we give a computation of `valid_mem_access` (for the last `read` instruction in our example program):

$$\begin{aligned} &\texttt{valid_mem_access}(m_7, q_0, 4) \\ \equiv\ & (p \neq \texttt{NULL} \wedge (p \leq q_0 \leq p+4)) \vee \\ & (c_1 \wedge q \neq \texttt{NULL} \wedge (q \leq q_0 \leq q+4)) \\ \equiv\ & (p \neq \texttt{NULL} \wedge (c_1 \Rightarrow p \leq q \leq p+4)) \vee \\ & (c_1 \wedge q \neq \texttt{NULL}) \end{aligned}$$

To obtain the second equivalence (which is mainly for illustration purposes on expression simplification) we have used equality reasoning (using the equation $q_0 = (c_1 ? q : p)$), if-lifting and Boolean simplification. Assuming non-failing `malloc`s, the formula can be simplified further, as we then have $(c_1 \Rightarrow p \leq q \leq p+4) \vee c_1$, which is equivalent to `true` and thus shows that this `read` instruction never produces an invalid memory access error.

The result of this transformation on our example—under the assumption of non-failing `malloc`s—is depicted in Fig. 7 in the middle.

2) Valid Frees and No Double Frees: A `free` is only valid if the address passed to it has been allocated in a prior `malloc` instruction and the memory region has not been deallocated before. Similar to the check for valid reads and writes, we first add `assume` statements with `malloc_assumptions`. Then we add $\texttt{assert}(\texttt{valid_free}(m, p))$ before each $m' = \texttt{free}(m, p)$ instruction.

3) No Memory Leaks: For simplicity, we assume that only one `return` statement is present in the LLVM code. This can always be ensured by using an optimization pass already available in LLVM. The last memory state before the return statement is marked as `finalmem`. For this memory state m_f we add the assertion $\texttt{assert}(\texttt{no_memory_leaks}(m_f))$. As for the other tests, the memory allocation assumptions (`malloc_assumption`) also have to be added.

4) Encoding Assumes and Asserts: The final step in our translation into the logic of bit-vector and arrays is an encoding of `assumes` and `asserts`, which also eliminates the execution conditions for basic blocks.

In general, the formula we want to check for validity (expressing that the program contains no memory error) has the form

$$\texttt{Pre} \wedge \texttt{Prog} \implies \texttt{Post}$$

Here `Pre` encodes the pre-conditions (i.e. the `assumes`), `Prog` is an encoding of the LLVM assembler program instructions, and `Post` are the post-conditions, i.e. a conjunction of the `asserts`.

This formula has to be negated since SMT solvers check for satisfiability instead of validity, resulting in

$$\texttt{Pre} \wedge \texttt{Prog} \wedge \neg\texttt{Post}$$

as the input formula for the SMT solver.

For pre- and post-conditions we have to add the execution conditions of the basic blocks they belong to, as these conditions only have to hold when their respective basic block is executed. This results in the formula $c_{\texttt{exec}}(I) \implies F_\alpha$ for an $\texttt{assume}(F_\alpha)$ statement with execution condition $c_{\texttt{exec}}(I)$. For each $\texttt{assert}(F_\omega)$ statement (with execution condition $c_{\texttt{exec}}(J)$) we introduce a new Boolean (1-bit) variable a_i together with a definition $a_i = (c_{\texttt{exec}}(J) \implies F_\omega)$ to the formula we want to check for satisfiability. Finally, we add the constraint $\neg(\bigwedge_i a_i)$ to our formula. This completes the transformation of LLVM programs to the logic of bit-vectors and arrays. Notice that after we have introduced the memory consistency constraints, the array accesses (read/write) have the usual array semantics, and no additional, "hidden" assumptions on memory access validity remain.

The result of this transformation on our example program is shown in Fig. 7 on the right.

B. Complexity

In this section we analyze the complexity of the memory consistency checks by giving size constraints on the formulas generated for each test. The size mainly depends on the number of memory instructions that occur in the program. For a rough estimate of the worst case complexity we assume that there are R `read` or `write` instructions in the program, M memory allocation instructions (`malloc`), and F `free` instructions.

For the memory consistency tests the complexity then is as follows:

- Valid reads and writes: $\mathcal{O}(R \cdot M \cdot F + M^2 \cdot F)$, as for each read/write access the formula `valid_mem_access`, and for each `malloc` the formula `malloc_assumption` has to be generated.
- Valid frees and no double-frees: $\mathcal{O}(F^2 \cdot M + M^2 \cdot F)$, as for each free instruction the formulas `valid_free` and `malloc_assumption` have to be generated.
- No memory leaks: $\mathcal{O}(M^2 \cdot F)$, as both the formulas `no_memory_leaks` and `malloc_assumption` have to be generated, and the size of `malloc_assumption` dominates the size of `no_memory_leaks`.

As typically the number of `malloc` and `free` instructions make only a small fraction of all program instructions, the complexity is acceptable for many practical cases. Under the assumption that the number of allocations/frees is bounded by a constant, the complexities even become linear (for the test on valid reads and writes) or constant (for all other tests).

IV. Optimizations

The encoding presented in the last section can be optimized in several ways, partly without imposing restrictions, and in part with slight additional assumptions.

A. Optimizing Memory Overlapping Constraints

As these constraints (having worst-case complexity $\mathcal{O}(M^2 \cdot F)$) dominate the size of the encoding for many checks, optimizing them has a huge potential for improving scalability.

One way to do this is to map allocated memory regions to fixed addresses such that the non-overlapping property is guaranteed by construction and non-overlapping-constraints are not needed at all.[4] This can be done under two assumptions:

1) No comparisons between pointers of different memory regions are made. (If such comparisons do occur, they should always evaluate to false.)

[4]Keeping track of the allocation status of the memory regions can be accomplished, e.g., by using an additional bit-array.

2) Memory space is sufficient to allow allocation of all memory regions “in advance”.

The first assumption can be ascertained by keeping track of the memory block(s) each pointer can reference, and generating suitable constraints on each pointer comparison. The second assumption holds for many programs, as long as they do not frequently allocate/deallocate large memory blocks. Notice that with a 64-bit memory architecture such an adversary situation practically never occurs.

B. Tracking Memory Regions for Pointers

In LLVM assembler, pointers are modified mainly using the `getelementptr` instruction, where offsets are added to existing pointers. This makes it possible to trace back the origin (memory region) of a pointer.[5] Notice that `phi` instructions may introduce multiple origins for a pointer.

Based on the origins of a pointer, the big disjunctions and conjunctions of the memory predicates (Fig. 4) can be simplified, by taking only pointers with a non-empty intersection with the set of origins of the test pointer into account.

The set of origins of a pointer can also be used to reduce the number of (Ackermann) constraints for functional consistency that are needed for the logic encoding of the memory array within the SMT solver.

C. Rewriting and Simple Decision Procedures

By using rewriting (substitution in equations) and simple decision procedures (linear arithmetic, if-lifting, Boolean simplification), many memory consistency checks can be handled in advance without having to invoke the SMT solver (see the Appendix for an example of such a simplification).

V. Related Work

Bounded model checking of C programs has been considered before. The well-known tool CBMC [6] reduces the problem to propositional logic instead of the logic of bit-vectors and arrays. Therefore, the formulas that need to be checked by a SAT solver have a much bigger size than the formulas that need to be checked by an SMT solver. Similarly, the tool F-Soft [14] generates formulas that are checked by a SAT solver. The use of SMT solvers for bounded model checking of C programs has been investigated in [1]. According to their evaluation, the use of SMT solvers typically outperforms the use of SAT solvers.

In these papers ([6], [14], [1]) no details are given on the employed memory models, however. On the other hand, several low-level memory models[6] for C-like languages have been proposed in the past ([19], [16], [7], [5], [13]). None of them deals with the combination of automated verification using bounded model checking and low-level compiler intermediate languages. Furthermore, they do not emphasize memory protection (or ignore it completely).

Tuch *at al.* [19], [18] discuss a typed memory model in the context of interactive theorem proving with the proof assistant Isabelle/HOL. It is shown that this typed memory model is sound with respect to the untyped memory model assumed by C.

The memory model presented by Leroy and Blazy [16] is similar to our model and considers `load`, `store`, `malloc`, and `free` instructions. While the disjointness of memory blocks allocated by separate `malloc` instructions is guaranteed, no such separation for accesses performed within the same memory block is ensured (e.g. accesses to different members of a structure). Leroy and Blazy prove properties of their memory model using the proof assistant Coq (such as semantic preservation of compiler passes).

Cohen *et al.* [7] introduce a typed memory model similar to [19] for a C-like toy programming language and show that this typed memory model is sound with respect to the untyped memory model assumed by C. They support pointer arithmetic and memory access (`load` and `store` instructions) at arbitrary locations in the memory, but do not consider memory protection (`malloc` and `free` instructions).

Gast [13] gives a formalism for reasoning about memory layouts of C programs. Proof obligations are formulated in Hoare logic and verified using Isabelle/HOL.

The memory model used in HAVOC is presented in [5]. HAVOC uses a reachability predicate based on the memory model in order to reason about heap-based data structures, but does not support memory protection.

Approaches using an intermediate language, like LLVM in our case, have also been proposed before, often in the realm of Java program verification, where Java Bytecode serves as a standardized, well-defined language (see, e.g., [20]). Another example is Microsoft’s BoogiePL which is used as an intermediate language for verifying Spec# programs [3].

VI. Conclusion

We have presented an approach for low-level bounded model checking of C programs that is based on the LLVM compiler front end and makes use of an SMT solver. Thus we are able to start the logical encoding not on the C source code level, but on the compiler

[5] Type casting arbitrary integers to pointers may undermine this possibility. Checks for such “invalid” casts would need to be added.

[6] In a low-level memory model the memory is not much more than an array of bytes and suitable disjointness or consistency conditions are stated explicitly.

intermediate language instead. This simplifies a mapping to the input language of an SMT solver (the logic of bit-vectors and arrays) considerably and results in a precise and simple modeling for all of C's programming constructs.

In this paper we have concentrated on formalizing memory consistency constraints and have given encodings for valid reads and writes, valid frees, and avoidance of double frees and memory leaks. Checking such constraints is important for finding residual bugs like, e.g., stack or heap buffer overflows, which are the predominant cause of security vulnerabilities.[7]

We are currently implementing the method proposed in this paper in the tool *LLBMC* which uses the SMT solver *Boolector* [4] as a back-end solver. First empirical results obtained on various test cases are quite encouraging, e.g., the double-free bug contained in the C program described in the Appendix is discovered within fractions of a second.

APPENDIX

In this appendix we give an example of how our method works on a small C program. The input program is shown in Fig. 5, which, in a first step, is translated by the LLVM front-end to abstract assembler. The LLVM code is shown in Fig. 6.

```
1   struct S {
2       int x;
3       struct S *n;
4   };
5
6   int main(int argc, char *argv[]) {
7       struct S *p, *q;
8
9       p = malloc(sizeof(struct S));
10      p->x = 5;
11      p->n = NULL;
12
13      if (argc > 1) {
14          q = malloc(sizeof(struct S));
15          q->x = 5;
16          q->n = p;
17      } else {
18          q = p;
19      }
20
21      __llbmc_assert(p->x + q->x == 10);
22
23      free(q);
24      free(p);
25
26      return 0;
27  }
```

Fig. 5. C program that is to be checked for correct memory accesses.

The LLVM code consists of a type definition section, followed by the translation of the `main` function. This translation was done for a 32-bit architecture, thus

[7]According to the *Vulnerability Type Distributions in CVE* Report of CVE-MITRE, 2007 (http://cwe.mitre.org/documents/vuln-trends/vuln-trends.pdf).

```
%struct.S = type { i32, %struct.S* }

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = call i8* @malloc(i32 8)
  %p = bitcast i8* %0 to %struct.S*
  %px = getelementptr %struct.S* %p, i32 0, i32 0
  store i32 5, i32* %px
  %pn = getelementptr %struct.S* %p, i32 0, i32 1
  store %struct.S* null, %struct.S** %pn
  %c1 = icmp sgt i32 %argc, 1
  br i1 %c1, label %if.then, label %if.end
if.then:
  %1 = call i8* @malloc(i32 8)
  %q = bitcast i8* %1 to %struct.S*
  %qx = getelementptr %struct.S* %q, i32 0, i32 0
  store i32 5, i32* %qx
  %qn = getelementptr %struct.S* %q, i32 0, i32 1
  store %struct.S* %p, %struct.S** %qn
  br label %if.end
if.end:
  %q0 = phi %struct.S* [%q, %if.then], [%p, %entry]
  %q0x = getelementptr %struct.S* %q0, i32 0, i32 0
  %2 = load i32* %px
  %3 = load i32* %q0x
  %4 = add i32 %2, %3
  %c2 = icmp eq i32 %4, 10
  call void @__llbmc_assert(i32 %c2)
  %5 = bitcast %struct.S* %q0 to i8*
  call void @free(i8* %5)
  %6 = bitcast %struct.S* %p to i8*
  call void @free(i8* %p)
  ret i32 0
}
```

Fig. 6. LLVM's abstract assembler (intermediate representation) code for the example C program of Fig. 5.

integers are 32-bit. The code for the `main` function consists of three basic blocks, where the assignment $q = p$ from the `else`-branch of the C program (lines 17–19) has been moved to the `phi` instruction.

Different steps in the translation of the LLVM intermediate code to a formula in the logic of bit-vectors and arrays, where memory consistency constraints have also been added, are shown in Fig. 7. On the left, basic blocks are still present (and annotated with execution conditions), but branches (or jumps) have been removed; the memory model's constraints are "hidden" in the memory state type `mem`. In the middle, memory constraints (via `assume` and `assert` statements) checking validity of all reads, writes, and frees have been added (according to Sec. III-A, where memory leak checks have been omitted in order to simplify presentation); the basic block structure is still present. On the right, `assume`s and `assert`s have been encoded (resulting in eight new, Boolean assertion variables $a_1, \dots, a_8$); in this formula `reads` and `writes` have ordinary array semantics, and thus it can be passed to an SMT solver.

Using simplification as in Sec. IV-C, all memory read and write assertions can be shown to hold and only the simplified assertion for the second `free` and the initial assertion, given in the C program in line 21, remain (see Fig. 8). This formula is satisfiable, and setting `argc` to 1 reveals the double free bug contained in the C program.

```
entry: [c_exec: T]
  mem m0 = initialmem()
  ptr p = nondet()
  mem m1 = malloc(m0, p, 8)
  mem m2 = write32(m1, p, 5)
  ptr pn = p + 4
  m3 = write32(m2, pn, NULL)
  bool c1 = (argc > 1)
if.then: [c_exec: c1]
  ptr q = nondet()
  mem m4 = malloc(m3, q, 8)
  mem m5 = write32(m4, q, 5)
  ptr qn = q + 4
  mem m6 = write32(m5, qn, p)
if.end: [c_exec: T]
  ptr q0 = (c1 ? q : p)
  mem m7 = (c1 ? m6 : m3)
  i32 px = read32(m7, p)
  i32 qx = read32(m7, q0)
  i32 sum = px + qx
  bool c2 = (sum == 10)
  assert(c2)
  mem m8 = free(m7, q0)
  mem m9 = free(m8, p)
  finalmem(m9)
  return 0
```

```
entry: [c_exec: T]
  assume(p >= MEM_MIN && p+7 <= MEM_MAX)
  assert(p <= p && p <= p+4)
  mem m2 = write32(m0, p, 5)
  ptr pn = p + 4
  assert(p <= pn && pn <= p+4)
  m3 = write32(m2, pn, NULL)
  bool c1 = (argc > 1)
if.then: [c_exec: c1]
  assume(q >= MEM_MIN && q+7 <= MEM_MAX &&
         (q+8 <= p || p+8 <= q))
  assert(p <= q && q <= p+4 ||
         c1 && q <= q && q <= q+4)
  mem m5 = write32(m3, q, 5)
  ptr qn = q + 4
  assert(p <= qn && qn <= p+4 ||
         c1 && q <= qn && qn <= q+4)
  mem m6 = write32(m5, qn, p)
if.end: [c_exec: T]
  ptr q0 = (c1 ? q : p)
  mem m7 = (c1 ? m6 : m3)
  assert(p <= p && p <= p+4 ||
         c1 && q <= p && p <= q+4)
  i32 px = read32(m7, p)
  assert(p <= q0 && q0 <= p+4 ||
         c1 && q <= q0 && q0 <= q+4)
  i32 qx = read32(m7, q0)
  i32 sum = px + qx
  bool c2 = (sum == 10)
  assert(c2)
  assert(q0 = p || (c1 && q0 = q))
  assert(p = p && p != q0 ||
         c1 && p = q && q != q0)
```

```
p >= MEM_MIN
p+7 <= MEM_MAX
a1 = (p <= p && p <= p+4)
m2 = write32(m0, p, 5)
pn = p + 4
a2 = (p <= pn && pn <= p+4)
m3 = write32(m2, pn, NULL)
c1 = (argc > 1)
c1 => q >= MEM_MIN
c1 => q+7 <= MEM_MAX
c1 => q+8 <= p || p+8 <= q
a3 = (c1 => (p <= q && q <= p+4 ||
      c1 && q <= q && q <= q+4))
m5 = write32(m3, q, 5)
qn = q + 4
a4 = (c1 => (p <= qn && qn <= p+4 ||
      c1 && q <= qn && qn <= q+4))
m6 = write32(m5, qn, p)
q0 = (c1 ? q : p)
m7 = (c1 ? m6 : m3)
a5 = (p <= p && p <= p+4 ||
      c1 && q <= p && p <= q+4)
px = read32(m7, p)
a6 = (p <= q0 && q0 <= p+4 ||
      c1 && q <= q0 && q0 <= q+4)
qx = read32(m7, q0)
sum = px + qx
c2 = (sum == 10)
a7 = (q0 = p || (c1 && q0 = q))
a8 = (p = p && p != q0 ||
      c1 && p = q && q != q0)
not(a1 && a2 && a3 && a4 &&
    a5 && a6 && a7 && a8 && c2)
```

Fig. 7. Different stages in the logical encoding of the program in Fig. 6. **Left:** Initial SMT encoding with memory states; basic blocks (and their execution conditions $c_{\texttt{exec}}$) are still present, as well as type annotations. **Middle:** SMT encoding with memory constraints (valid reads/writes). **Right:** Final formula that can be passed to an SMT solver for the logic of bit-vectors and arrays.

```
p >= MEM_MIN
p+7 <= MEM_MAX
m2 = write32(m0, p, 5)
m3 = write32(m2, p+4, NULL)
c1 = (argc > 1)
c1 => q >= MEM_MIN
c1 => q+7 <= MEM_MAX
c1 => q+8 <= p || p+8 <= q
m5 = write32(m3, q, 5)
m6 = write32(m5, q+4, p)
px = c1 ? read32(m6, p) : read32(m3, p)
qx = c1 ? read32(m6, q) : read32(m3, p)
p = q || argc <= 1 || px + qx != 10
```

Fig. 8. SMT formula from Fig. 7 (right) after equational rewriting, if-lifting, basic linear arithmetic and Boolean simplification. All memory consistency constraints for reads and writes could be removed without needing to invoke the SMT solver.

References

[1] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *STTT*, vol. 11, no. 1, pp. 69–83, 2009.

[2] D. Babić and A. J. Hu, "Structural abstraction of software verification conditions," in *Proc. CAV 2007*, ser. LNCS, vol. 4590, 2007, pp. 371–383.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Proc. FMCO 2005*, 2005, pp. 364–387.

[4] R. Brummayer and A. Biere, "Boolector: An efficient SMT solver for bit-vectors and arrays," in *Proc. TACAS 2009*, ser. LNCS, vol. 5505, 2009, pp. 174–177.

[5] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "A low-level memory model and an accompanying reachability predicate," *STTT*, vol. 11, no. 2, pp. 105–116, 2009.

[6] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. TACAS 2004*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[7] E. Cohen, M. Moskal, S. Tobies, and W. Schulte, "A precise yet efficient memory model for C," *ENTCS*, vol. 254, pp. 85–103, 2009.

[8] D. R. Cok, "Improved usability and performance of SMT solvers for debugging specifications," *STTT*, 2010.

[9] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer, "Unifying type checking and property checking for low-level code," in *Proc. POPL 2009*, 2009, pp. 302–314.

[10] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of bounded model checking at an industrial setting," in *Proc. CAV 2001*, ser. LNCS, vol. 2102, 2001, pp. 436–453.

[11] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," in *Proc. ASE 2009*, 2009, pp. 137–148.

[12] M. K. Ganai and A. Gupta, "Accelerating high-level bounded model checking," in *Proc. ICCAD 2006*, 2006, pp. 794–801.

[13] H. Gast, "Reasoning about memory layouts," in *Proc. FM 2009*, ser. LNCS, vol. 5850, 2009, pp. 628–643.

[14] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *TCS*, vol. 404, no. 3, pp. 256–274, 2008.

[15] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO 2004*, 2004, pp. 75–88.

[16] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *JAR*, vol. 41, no. 1, pp. 1–31, 2008.

[17] Y. Moy, "Automatic modular static safety checking for C programs," Ph.D. dissertation, Université Paris-Sud, France, 2009.

[18] H. Tuch, "Formal verification of C systems code: Structured types, separation logic and theorem proving," *JAR*, vol. 42, no. 2–4, pp. 125–187, 2009.

[19] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *Proc. POPL 2007*, 2007, pp. 97–108.

[20] W. Visser, K. Havelund, G. P. Brat, and S. Park, "Model checking programs," in *Proc. ASE 2000*, 2000, pp. 3–12.