# LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR*

Florian Merz, Stephan Falke, and Carsten Sinz

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

**Abstract.** Bounded model checking (BMC) of C and C++ programs is challenging due to the complex and intricate syntax and semantics of these programming languages. The BMC tool LLBMC presented in this paper thus uses the LLVM compiler framework in order to translate C and C++ programs into LLVM's intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. In contrast to many other tools, LLBMC uses a flat, bit-precise memory model. It can thus precisely model, e.g., memory-based re-interpret casts as used in C and static/dynamic casts as used in C++. An empirical evaluation shows that LLBMC compares favorably to the related BMC tools CBMC and ESBMC.

## 1 Introduction

Bounded model checking (BMC) [3], introduced by Biere *et al.* in 1999, is a popular technique for bug finding and verification of hardware designs that is widely used in an industrial setting. For bug finding of software, BMC of C programs was introduced by Clarke *et al.* in 2004 [8], and has shown its strength in checking a variety of aspects of embedded and low-level system software (see, e.g., [16, 23]). Tools implementing BMC for C programs include CBMC [8] (developed by D. Kröning *et al.*), F-Soft [15] (developed at NEC Laboratories America), SMT-CBMC [1] (developed by A. Armando *et al.*), and ESBMC [10] (developed by L. Cordeiro *et al.*).

To build a BMC tool that supports all language features of a high-level language like C or C++ reliably, including common non-standard extensions that are used by, e.g., the GCC compiler, is a daunting task. This is mostly due to the complex syntax and intricate, sometimes ambiguous, semantics of these languages. The bounded model checker LLBMC presented in this paper therefore performs BMC not on the source code level but on the level of a compiler intermediate representation (IR). This approach offers a range of advantages:

* 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.

– The compiler IR possesses a much simpler syntax and semantics than C/C++ and thus eases a logical encoding considerably. Furthermore, most features of C and C++ can be supported without much effort.[1]
– The program that is analyzed is much closer to the program that is actually executed on the computer since ambiguities of C/C++'s semantics have already been resolved. Furthermore, it becomes possible to find bugs introduced by the compiler.
– In producing the IR, compilers already use program optimizations that can also result in simplified BMC problems.
– The use of a compiler IR makes it possible to perform BMC on programs written in a variety of programming languages.

The use of an IR makes LLBMC, to the best of our knowledge, the *only* BMC tool that can be successfully applied to non-trivial C++ programs (CBMC contains rudimentary support for C++ but failed to analyze nearly all of the over 50 C++ programs we tried it on). A drawback of using an IR is that bugs that are (intuitively) present in the C/C++ program may be "optimized away" by the compiler (but notice that the bugs would then also not occur during execution of the program if the same compiler is used to produce the executable).

Besides using a compiler IR, LLBMC offers the following key features:

**Large set of built-in checks:** LLBMC provides a comprehensive set of built-in checks which are described in detail in Sect. 2.

**Extensive simplification:** LLBMC uses simplification techniques on different levels. First, using the optimizations of the compiler front-end generates smaller and simpler IR programs. In particular, memory-related compiler optimization techniques (e.g., moving memory operations to registers whenever possible) can simplify the BMC problem significantly. Second, rewriting techniques are used on the logical representation, e.g., to propagate constants or to simplify arithmetical and Boolean expressions.

**Memory is modeled as a flat byte-array:** This design decision is in contrast to what is implemented in many other tools (e.g., pre-3.9 versions of CBMC [8] or deductive verification tools such as VCC [9]) which use a typed memory model. In a typed memory model, memory is a collection of typed objects rather than a sequence of bytes. Using a byte-array makes it possible to support programs which make use of C's weak type system, e.g., by converting an `int` to a sequence of `char`s that are written to a file. Another example is the use of a `union` for the conversion between types. Typically, modeling memory on the byte level causes a performance penalty in BMC, but LLBMC uses simplification techniques that, according to an empirical evaluation, compensate for this.

Section 2 recalls BMC of software and discusses the built-in checks of LLBMC. The compiler framework LLVM is briefly introduced in Sect. 3, while Sect. 4 and 5 give details on LLBMC's approach. An empirical evaluation is presented in Sect. 6. Section 7 discusses related work and Sect. 8 concludes.

[1] Currently, LLBMC does not support floating-point numbers, exception handling and run-time type information (RTTI).

## 2 BMC and LLBMC's Built-In Checks

Software inherently deals with unbounded data structures such as linked lists or trees. This may give rise to infinite program runs and property checking of such programs is in general undecidable. For bug finding, BMC thus limits all program runs to finite ones, thereby achieving decidability. The bound is imposed by restricting the number of nested function calls and loop iterations that are considered. BMC performs function inlining and loop unrolling (up to these bounds), resulting in one large function that is then subject to further analysis.

LLBMC has an extensive set of built-in checks for commonly occurring bugs in C programs. Furthermore, user defined checks (specified via C's `assert` function) are supported as well. Each of these checks can be enabled or disabled independently, but most of them are enabled by default.

**Arithmetic overflow and underflow:** Arithmetic overflow[2] occurs when the result of a signed or unsigned arithmetic operation cannot be represented with the available number of bits. While the semantics of unsigned integer overflows is well-defined by the C standard using modular arithmetic, this is not true for their signed counterparts. The semantics of signed integer overflows are intentionally under-specified in the standard to give different implementations room for optimizations. Thus, any signed arithmetic overflow in a program may give rise to undefined behavior and LLBMC checks for them by default. Checks for unsigned arithmetic overflows are enabled only if requested.

**Logic or arithmetic shift exceeding the bit-width:** While most programmers are familiar with arithmetic overflows, shift operations are a less well-known cause for undefined behavior. The C standard leaves shift operations like $n << l$ undefined if $l$ is larger than or equal to the bit-width of $n$.[3] LLBMC supports checks for this kind of error by default since this behavior is not expected by most programmers.

**Memory access at invalid addresses:** One of the most important classes of errors is caused by invalid memory access operations. The prime example of this are security-critical buffer overflows. An access operation for an object on the heap is only valid if it is completely contained within a block of memory which was previously allocated using `malloc`. Due to C's unrestricted pointer arithmetic, invalid memory access operations are a frequent source of crashes and vulnerabilities. LLBMC detects invalid memory accesses on the stack, on the heap, and for global variables.

**Invalid memory allocation:** Heap memory allocations are considered invalid by LLBMC if a memory block of the requested size can be allocated under no circumstances. Currently, LLBMC approximates this by checking if the total size of all allocated blocks would exceed the size of the heap.

[2] In the following, "overflow" is used to denote both overflow and underflow.

[3] On many architectures, shifting $x$ by $l$ bits is equivalent to shifting $x$ by $l \bmod b$ bits, where $b$ is the bit-width of $x$'s data type.

**Invalid memory de-allocation:** A call to `free(p)`/`delete p` is invalid if a memory block starting at `p` was already de-allocated, was never allocated, or if `p` points to an address which is not the first byte of an allocated memory block. LLBMC checks whether either of these situations occurs.

**Overlapping memory regions in `memcpy`:** In C, `memcpy` is used to copy the content of a block of memory from one location to another. The result is undefined, though, if the source and destination blocks overlap. LLBMC checks that this does not happen.

**Memory leaks:** Memory leaks occur when blocks of memory are allocated, but never de-allocated. For long running programs this might cause an out-of-memory situation. LLBMC checks for memory leaks as described in [26].

**User defined assertions:** In addition to the built-in assertions, LLBMC supports checking user defined properties expressed in C via C's `assert` function or the LLBMC-specific `__llbmc_assert`.[4] Assumptions can also be specified using the built-in function `__llbmc_assume`.

**BMC specific assertions:** Finally, LLBMC is able to automatically detect insufficient bounds for nested function calls and loop iterations that cause BMC to be incomplete since not all programs executions are considered in these cases.

## 3 LLVM

LLBMC uses the LLVM compiler framework (versions from 2.7 through 3.0) and its intermediate representation LLVM-IR [18]. This makes it possible to use LLBMC on programs that are written in several programming languages, since compiler front-ends for, amongst others, C and C++, are available. The main target of LLBMC is bounded model checking of C programs, but C++ programs that do not use exception handling or run-time type information (RTTI) are also supported.

LLVM's intermediate representation is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers. A program in LLVM-IR consists of type definitions, global variable declarations, and the program itself, which is represented as a set of functions, each consisting of a graph of basic blocks. Each basic block in turn is a list of instructions, where the instruction set can broadly be split into six types:

1. Three-address-code (TAC) instructions working on registers or constants.
2. The memory access instructions `load` and `store`.
3. Address calculations using `getelementptr`.
4. Conditional and unconditional branch instructions, `phi` instructions.
5. Function call instructions.
6. Bit-level instructions like extensions, truncations, and type casts.

Here, (conditional and unconditional) branch instructions are only allowed as the last instruction of a basic block. The branch instructions between basic blocks

[4] `__llbmc_assert` is used only for specification purposes and not checked at runtime.

induce a *basic block graph*, in which edges are annotated with the condition under which the transition between the two basic blocks is taken.

Programs in LLVM-IR are in *static single assignment (SSA)* form, i.e., each (scalar) variable is assigned exactly once in the static program. Assignments to scalar variables can thus be treated as logical equivalences. Due to its restricted instruction set, the use of SSA form, and its low-level nature, converting an LLVM-IR program into a logical representation is considerably easier than operating on the source code of a high-level programming language.

The simple C program given in Fig. 1 is used as a running example. This program is converted into the LLVM-IR program also shown in Fig. 1 by the C front-end `llvm-gcc` (on a 32-bit architecture using the optimization level `-O2`). Notice that the low-level bit-field and union operations have been replaced by word-level instructions by the front-end.

```
union U {
    char c[4];
    struct { int v: 31; int s: 1; } t;
    int i;
};

void __llbmc_main(char n) {
    union U *u; char *p; int i;
    u = malloc(sizeof(union U));
    p = u->c;
    u->t.s = 1;
    u->t.v = 0;
    p[0] = n;
    __llbmc_assert(u->i == INT_MIN);
}
```

```
define void @__llbmc_main(i8 %n) {
entry:
  %0 = call i8* @malloc(i32 4)          ; u = malloc(sizeof(
  %1 = bitcast i8* %0 to i32*           ;             union U));
  store i32 -2147483648, i32* %1        ; u->t.s = 1; u->t.v = 0;
  store i8 %n, i8* %0                   ; p[0] = n;
  %2 = load i32* %1                     ; u->i
  %3 = icmp eq i32 %2, -2147483648      ;       == INT_MIN ?
  %4 = zext i1 %3 to i32
  call void @__llbmc_assert(i32 %4)
  ret void
}
```

**Fig. 1.** Example C program. It is converted into the given LLVM-IR program by the C front-end `llvm-gcc`. The function `__llbmc_main` is taken as the starting point for BMC.

## 4 The Approach of LLBMC

The overall approach of LLBMC is as follows: First, an LLVM compiler front-end (such as `clang` or `llvm-gcc`) is used in order to convert a C program into an LLVM-IR program. This LLVM-IR program is then converted into LLBMC's internal logical representation ILR. The ILR formula is simplified by LLBMC using rewrite rules before being passed to an SMT solver. If the SMT solver finds a satisfying assignment (corresponding to a bug in the program), this can be converted into a counterexample, first on the ILR level and then on the LLVM-IR level. The approach is summarized in Fig. 2.

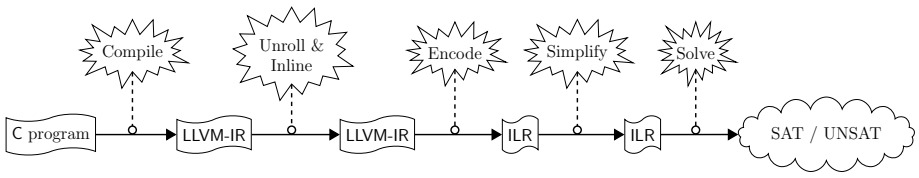


**Fig. 2.** LLBMC's approach.

### 4.1 From LLVM-IR to ILR

After parsing the LLVM-IR program, a number of transformations are applied to it (e.g., loops are unrolled and functions are inlined a fixed number of times and the control flow graph is simplified).[5] The transformed program is then converted into ILR, which is a representation of a formula in the logic of bit-vectors and arrays with some extensions that, e.g., handle the special semantics of memory allocation instructions like `malloc` and `free`. This format closely follows LLVM's instruction set, but differs from LLVM-IR in that it provides an explicit state object for the memory content as well as for the state of the memory allocation system. These state objects encode the dependencies between memory access instructions and `malloc`/`free`, respectively. With an explicit representation of the memory state, dependencies between memory-related instructions in LLVM (which were implicitly given by the ordering of the operations) are made explicit in the ILR formula. This makes the expressions in ILR order-independent.

[5] LLBMC accepts arbitrary LLVM-IR programs as input and does not depend on any optimizations performed by the compiler. For efficiency reasons, LLBMC internally runs LLVM's `mem2reg` pass in order to promote stack memory to registers when possible. Furthermore, the `indvars` pass is used in order to automatically determine the (static) number of loop iterations for certain kinds of simple loops.

Translation of LLVM's three-address-code, memory access, address calculation, and bit-level instructions is straightforward, since these instructions are part of the theory of bit-vectors and arrays—or can easily be encoded into it.

`phi` instructions are a common tool in compiler IRs that use SSA form. They are used to select the correct value for a variable from a set of previous values (e.g., when control flow merges after an *if-then-else* statement). In general, a `phi` expression in ILR has the form

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

where the value that the variable $i'$ takes is one 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, i.e., the value of $i'$ is always uniquely determined.

For SMT solvers, a `phi` expression can be translated into a sequence of ITE (if-then-else) operators (written in C syntax below):

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

The conditions $c_j$ are not given explicitly on the LLVM-IR level, though. Instead, basic blocks are used as designators. These basic blocks refer to the immediate predecessor in the basic block graph from which the current basic block has been reached. It thus becomes necessary to compute the conditions $c_j$. This is accomplished as follows. An *execution condition* $c_{\mathsf{exec}}(b)$ is associated with each basic block $b$. Execution conditions can be calculated recursively. Let $P(b)$ denote the set of predecessors of $b$ in the basic block graph, and let $t(b, b')$ be the condition under which the transition from basic block $b$ to $b'$ is taken (the edge label in the basic block graph). Then

$$c_{\mathsf{exec}}(b) = \bigvee_{b' \in P(b)} \left( c_{\mathsf{exec}}(b') \wedge t(b', b) \right)$$

if $P(b) \neq \emptyset$, and $c_{\mathsf{exec}}(b) = \top$ otherwise. Then, the basic block $b'$ in a `phi` instruction on the LLVM-IR level that occurs in the basic block $b$ can be replaced by the condition $c_{\mathsf{exec}}(b') \wedge t(b', b)$ on the ILR level.

Notice that each $c_{\mathsf{exec}}(b)$ requires only linear space in the number of predecessors of the basic block $b$ if the recursive definition is not expanded but encoded by introducing new Boolean variables for each $c_{\mathsf{exec}}(b)$ and $t(b, b')$ instead.

### 4.2 Adding checks to the ILR formula

After the initial ILR formula has been generated, it is annotated with `LLBMC`'s built-in checks. Most of these checks are supported by a predicate that is part of ILR, e.g., there are `no_overflow`, `valid_access`, and `valid_free` predicates. Then, an instruction that can possibly overflow is guarded by an assertion that no overflow occurs, a memory access instruction is guarded by an assertion that the access is valid, and so on.

After converting the LLVM-IR program from Fig. 1 into ILR and adding the predicates for the built-in checks, the ILR formula shown in Fig. 3 is obtained. Here, assertions are encoded in such a way that only the first error in the program is reported.

```
i8 %n = nondef()
i8* %0 = nondef()
heap %1 = malloc(%initialHeap, %0, i32__4)
bool %2 = valid_malloc(%initialHeap, %0, i32__4)
assert(%2, "valid_malloc")
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32__2147483648)
bool %5 = valid_access(%1, %3, i32__4)
bool %6 = and(%2, %5)
bool %7 = not(%2)
bool %8 = or(%7, %6)
assert(%8, "valid_store")
mem %9 = store(%4, %0, %n)
bool %10 = valid_access(%1, %0, i32__1)
bool %11 = and(%6, %10)
bool %12 = not(%6)
bool %13 = or(%12, %11)
assert(%13, "valid_store")
i32 %14 = load(%9, %3)
bool %15 = valid_access(%1, %3, i32__4)
bool %16 = and(%11, %15)
bool %17 = not(%11)
bool %18 = or(%17, %16)
assert( valid_load , %18)
bool %19 = compare(EQ, %14, i32__2147483648)
bool %20 = and(%16, %19)
bool %21 = not(%16)
bool %22 = or(%21, %20)
assert(%22, "custom")
```

**Fig. 3.** ILR formula obtained for the LLVM-IR program from Fig. 1.

### 4.3 Simplification of the ILR formula

Similar to [25], `LLBMC` uses term rewriting in order to simplify the ILR formula before passing it to an SMT solver (`LLBMC` uses `Boolector` [4] by default, but also supports `STP` [13] and `Z3` [22]). Most of the rewrite rules used by `LLBMC` are rather simple and correspond to constant propagation or simple arithmetical and logical properties. In total, approximately 150 (conditional) rewrite rules have been implemented in `LLBMC` in order to simplify the ILR formula.

Before the ILR formula is passed to the SMT solver, ILR's predicates for built-in checks are expanded if they are not already supported by the SMT solver:

- Arithmetic overflow detection is supported by many current SMT solvers. Otherwise, it can be encoded in bit-vector logic directly (see, e.g., [5]).
- Checks for logic and arithmetic shift exceeding the bit-width can easily be encoded in bit-vector logic using suitable comparison expressions. The same is true for invalid memory allocations (i.e., memory allocations that are "too big") and overlapping memory regions in `memcpy`.
- Invalid memory access, invalid `free`, and memory leak detection is more complex. Their encoding is discussed in detail in Sect. 5.

After expanding the predicates for the built-in checks and rewrite-based simplifications of the formula from Fig. 3, the formula shown in Fig. 4 is obtained.

```
i8  %n = nondef()
i8* %0 = nondef()
i32* %3 = bitcast(%0)
mem %4 = store(%initialMemory, %3, i32__2147483648)
mem %9 = store(%4, %0, %n)
i32  %14 = load(%9, %3)
bool %19 = compare(EQ, %14, i32__2147483648)
assert(%19, "custom")
```

**Fig. 4.** ILR formula obtained by simplifying the ILR formula from Fig. 3.

### 4.4 Counterexample generation

The simplified ILR formula is then passed to an SMT solver for the logic of bit-vectors and arrays. If the formula is satisfiable, any satisfying assignment corresponds to a bug in the program. By mapping ILR variables to the corresponding instructions in the LLVM-IR program and simulating execution with these values, a trace of the LLVM-IR program that exhibits the bug can be obtained. The bug exhibited by assigning $-128$ to `n` (and where `malloc` returns the address `0x7ffffffc`) in the running example is displayed in Fig. 5.

## 5 Encoding Memory Checks

In this section it is described how the memory-related check predicates are expanded into formulas that can be handled by current SMT solvers. The following discussion only considers the heap. Memory blocks on the heap are allocated using `malloc` and de-allocated using `free`. In ILR, these functions take the form

$$h' = \texttt{malloc}(h, p, s)$$

```
define void @__llbmc_main(i8 %n) {   ; i8 %n = -128
entry:                               ; executed
  %0 = call i8* @malloc(i32 4)       ; 0x7ffffffc
  %1 = bitcast i8* %0 to i32*        ; 0x7ffffffc
  store i32 -2147483648, i32* %1
    ; [0x7ffffffc] -> [0x00 0x00 0x00 0x80]
  store i8 %n, i8* %0
    ; [0x7ffffffc] -> [0x80 0x00 0x00 0x80]
  %2 = load i32* %1                  ; -2147483520
  %3 = icmp eq i32 %2, -2147483648   ; 0
  %4 = zext i1 %3 to i32             ; 0
  call void @__llbmc_assert(i32 %4)  ; FAILED
}
```

**Fig. 5.** Error trace exhibiting a bug in the running example.

$$h' = \texttt{free}(h, p)$$

where $h$, $h'$ are (explicit but abstract) heap allocation states, $p$ is a pointer, and $s$ is the size (in bytes) of the memory block that is to be allocated by `malloc`. Notice that `malloc` takes the pointer $p$ as a parameter and does not provide it as a return value. In the conversion from LLVM-IR to ILR, `malloc` is always preceded by a new pointer variable declaration for $p$, and `malloc` intuitively adds suitable constraints on this pointer. The heap allocation state $h'$ returned by `malloc` can then be considered as having these constraints added. The `free` function modifies the heap allocation state in such a way that the (currently allocated) memory block starting at address $p$ is de-allocated.

`LLBMC` supports two different encodings for the memory checks: a "global" encoding (following [26]) and a "local" encoding (following [12]). In the global encoding, the memory check predicates are expanded by taking the whole formula into consideration at once. In contrast, the local approach is based on conditional rewrite rules that only take the immediate arguments of the predicates into account. As an example, the expansion of the `valid-access` predicate is discussed below, the remaining memory-related check predicates are handled similarly, see [26, 12] for details.

The `valid-access` predicate has the form

$$\texttt{valid-access}(h, p, s)$$

where $h$ is a heap allocation state, $p$ is a pointer, and $s$ is the size (in bytes) of the memory block that is to be accessed. The intended semantics of this predicate is that it is true in exactly those cases where the memory region $[p, p + s)$ is contained within a memory block that is currently allocated in $h$.

The "global" encoding of `valid-access` is given below. The encoding of $\texttt{valid-access}(h, p, s)$ iterates over all `malloc`s that potentially took place when obtaining the heap allocation state $h$. $\texttt{valid-access}(h, p, s)$ is then true if a

`malloc` that actually took place allocated a memory block that contains $[p, p+s)$ and if this memory block was not de-allocated since then.

$$\texttt{valid-access}(h, p, s) \quad \equiv \quad \bigvee_{\substack{h' \preceq h \\ I\colon h' = \texttt{malloc}(h'', q, t)}} \Big( c_{\mathsf{exec}}(I) \ \wedge \ q \leq p \ \wedge \ p + s \leq q + t \ \wedge \ \neg \texttt{deallocated}(h', h, q) \Big)$$

$$\texttt{deallocated}(h, h', p) \quad \equiv \quad \bigvee_{\substack{h \preceq h^* \preceq h' \\ I\colon h^* = \texttt{free}(h'', q)}} \Big( c_{\mathsf{exec}}(I) \ \wedge \ p = q \Big)$$

Here, $c_{\mathsf{exec}}(I)$ is the execution condition of (the basic block containing the) instruction $I$. $h' \preceq h$ means that $h'$ is a (direct or indirect) predecessor of $h$ in the history of heap allocation states.

The "local" encoding of `valid-access` is given in the following. It uses conditional rewrite rules of the form $C \mid \ell \longrightarrow r$, expressing that $\ell$ can be rewritten to $r$ if the condition $C$ can be evaluated to true. A memory access in the "empty" heap allocation state $\varepsilon$ is never valid (first rewrite rule). An access within an allocated memory block is always valid (second rewrite rule), and, e.g., an access that partially overlaps with a memory block that is getting de-allocated is never valid (last rewrite rule).

$$\begin{aligned} \texttt{contained}(p, s, q, t) \ &:= \ p \leq q \ \wedge \ q + t \leq p + s \\ \texttt{disjoint}(p, s, q, t) \ &:= \ p + s \leq q \ \vee \ q + t \leq p \end{aligned}$$

$$\begin{aligned} & \texttt{valid-access}(\varepsilon, p, s) \\ & \longrightarrow \ \bot \\ \texttt{contained}(p, s, q, t) \mid \ & \texttt{valid-access}(\texttt{malloc}(h, p, s), q, t) \\ & \longrightarrow \ \top \\ \neg\texttt{contained}(p, s, q, t) \mid \ & \texttt{valid-access}(\texttt{malloc}(h, p, s), q, t) \\ & \longrightarrow \ \texttt{valid-access}(h, q, t) \\ \neg\texttt{valid-free}(h, p) \mid \ & \texttt{valid-access}(\texttt{free}(h, p), q, t) \\ & \longrightarrow \ \texttt{valid-access}(h, q, t) \\ \texttt{valid-free}(h, p) \wedge \texttt{disjoint}(p, \texttt{bsize}(h, p), q, t) \mid \ & \texttt{valid-access}(\texttt{free}(h, p), q, t) \\ & \longrightarrow \ \texttt{valid-access}(h, q, t) \\ \texttt{valid-free}(h, p) \wedge \neg\texttt{disjoint}(p, \texttt{bsize}(h, p), q, t) \mid \ & \texttt{valid-access}(\texttt{free}(h, p), q, t) \\ & \longrightarrow \ \bot \end{aligned}$$

Here, `valid-free` determines whether a `free` is valid, i.e., whether it changes the heap allocation state. `bsize` determines the size of the (currently allocated) memory block beginning at $p$. See [12] for details on their encodings.

## 6 Evaluation

In order to evaluate `LLBMC`'s performance, we compared it with two other BMC tools: the `C` Bounded Model Checker `CBMC` [8] and the Efficient SMT-Based

Context-Bounded Model Checker ESBMC 1.16 [10].[6] CBMC 3.9 contains significant changes concerning the memory model since the typed memory model has been replaced by a (mostly) byte-oriented model [17]. Since this new memory model is less mature than CBMC's typed memory model, we also included the previous version of CBMC (3.8) in the comparison.

Benchmarks for the comparison were selected from a variety of papers and sources in order to minimize any kind of bias. In total, 175 C programs were included. The benchmark selection did not include any C++ programs since ESBMC does not support C++ and CBMC's C++ support is still very rudimentary. We have, however, successfully used LLBMC on 57 C++ programs containing advanced features such as multiple inheritance, STL containers, and templates.[7] All four examples presented in [21] and all benchmarks mentioned in [1] were included in the evaluation. All of the benchmarks from the NEC Laboratories America benchmark suite[8] were considered, but only those without infinite loops were chosen (otherwise BMC is incomplete for all loop unrolling depths). A family of four benchmarks implementing queues was constructed by us. Eight examples provided in SLAyer's web interface[9] were included, as well as all examples distributed with the Static Modular Assertion ChecKer SMACK[10] [24], except for those where non-trivial loop invariants were used (which are not supported by either of the evaluated tools). Ten examples from the URBiVA distribution [20] were added as well. Finally, two sets of worst-case execution time benchmark suites were added to the selection of benchmarks: the SNU[11] and the WCET[12] [14] suites. The complete benchmark collection and LLBMC itself are available at http://baldur.iti.kit.edu/llbmc/.

In order to compensate for different default settings, CBMC was run with the options --bounds-check, --div-by-zero-check, --pointer-check, and --overflow-check and ESBMC was run with the option --overflow-check. Otherwise, CBMC and ESBMC where run with their default settings, in particular concerning the choice of SAT or SMT solvers. For LLBMC, the C programs were converted to LLVM-IR using llvm-gcc (version 2.8) with all compiler optimizations switched off. Furthermore, LLBMC was configured to use Boolector as its SMT solver. The loop unrolling and function inlining bounds were set to the lowest possible values to detect a bug or show that no bug is present.

The evaluation was performed on an Intel® Core™ 2 Duo machine with 2.4GHz running Ubuntu Linux 11.04. For each benchmark, the memory limit was set to 2.5GB and the time limit was set to 15 minutes. The results of the comparison are shown in Table 1.

[6] F-Soft [15] and SMT-CBMC [1] are not publicly available.

[7] For these C++ programs, CBMC 3.8 correctly solves nine programs, fails to handle 41 programs, and produces incorrect results for seven programs. For CBMC 3.9, the numbers are four, 53, and zero, respectively.

[8] http://www.nec-labs.com/research/system/systems_SAV-website/

[9] http://rise4fun.com/SLAyer

[10] http://www.zvonimir.info/projects/

[11] http://www.cprover.org/satabs/examples/SNU_Real_Time_Benchmarks/

[12] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

| Benchmark | | LLBMC | | | | CBMC 3.8 | | | | CBMC 3.9 | | | | ESBMC 1.16 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Family | N | S | O | F | I | S | O | F | I | S | O | F | I | S | O | F | I |
| [21] | 4 | **4** | 0 | 0 | 0 | **4** | 0 | 0 | 0 | 2 | 0 | 2 | 0 | 1 | 1 | 2 | 0 |
| [1] | 32 | **32** | 0 | 0 | 0 | 28 | 4 | 0 | 0 | 28 | 4 | 0 | 0 | 31 | 0 | 1 | 0 |
| NECLA | 45 | **44** | 0 | 0 | 1 | 34 | 4 | 5 | 2 | 29 | 2 | 9 | 5 | 31 | 4 | 6 | 4 |
| Queue | 4 | **4** | 0 | 0 | 0 | 3 | 1 | 0 | 0 | 3 | 1 | 0 | 0 | 3 | 1 | 0 | 0 |
| SLAyer | 8 | **8** | 0 | 0 | 0 | 5 | 0 | 0 | 3 | 4 | 0 | 0 | 4 | 4 | 0 | 1 | 3 |
| SMACK | 38 | **38** | 0 | 0 | 0 | 30 | 3 | 0 | 5 | 16 | 0 | 0 | 22 | 31 | 0 | 0 | 7 |
| SNU | 6 | **6** | 0 | 0 | 0 | 5 | 0 | 1 | 0 | 5 | 0 | 1 | 0 | **6** | 0 | 0 | 0 |
| URBiVA | 10 | **10** | 0 | 0 | 0 | 9 | 0 | 0 | 1 | 5 | 0 | 0 | 5 | 4 | 1 | 5 | 0 |
| WCET | 28 | 26 | 1 | 0 | 1 | **27** | 1 | 0 | 0 | **27** | 1 | 0 | 0 | 26 | 1 | 0 | 1 |
| Total | 175 | **172** | 1 | 0 | 2 | 145 | 13 | 6 | 11 | 119 | 8 | 12 | 36 | 137 | 8 | 15 | 15 |
| % | | **98.3** | 0.6 | 0.0 | 1.1 | 82.9 | 7.4 | 3.4 | 6.3 | 68.0 | 4.6 | 6.9 | 20.6 | 78.3 | 4.6 | 8.6 | 8.6 |

**Table 1.** Results of the evaluation. "N" denotes the number of instances in a benchmark family. "S" denotes the number of successfully solved instances (correctly detected bugs or absence of bugs proved), "O" the number of times the tool ran out of time or memory, "F" the number of failures to handle the input program, and "I" the number of incorrect results (i.e., the tool reports a non-existing "bug" or misses a bug).

Notice that LLBMC is able to successfully solve (i.e., find bugs in) over 18% more benchmarks than the best other tool in the comparison. The evaluation, however, contains two incorrect results reported by LLBMC:

- In the benchmark family WCET: in the benchmark containing Duff's device (`duff.c`), a loop is not recognized as such by LLVM. Because of this, LLBMC incorrectly reports insufficient loop unrolling bounds.
- In the benchmark family NECLA: LLVM's optimizations (even using the compiler setting `-O0`) cause information about signedness of an arithmetic operation to be lost. LLBMC then does not check the operation for signed arithmetic overflow and misses an overflow bug.

Notice that both CBMC and ESBMC have a significantly larger number of incorrect results, i.e., report more non-existing "bugs" or miss bugs.

The cactus plot in Fig. 6 compares the run-times of the four tools. Benchmarks that could not be handled or where an incorrect result was reported are considered as time-outs. The plot clearly shows that LLBMC produces more correct results in a shorter amount of time than any of the competing tools. Also notice the decrease in the number of correct results between CBMC 3.8 and 3.9.

## 7 Related Work

Bounded model checking of hardware was introduced by Biere *et al.* in 1999 [3] as an alternative to symbolic model checking using binary decision diagrams (BDDs) [6]. In 2004 Clarke *et al.* were the first to describe the application of BMC to software (more specifically, C programs) [8].

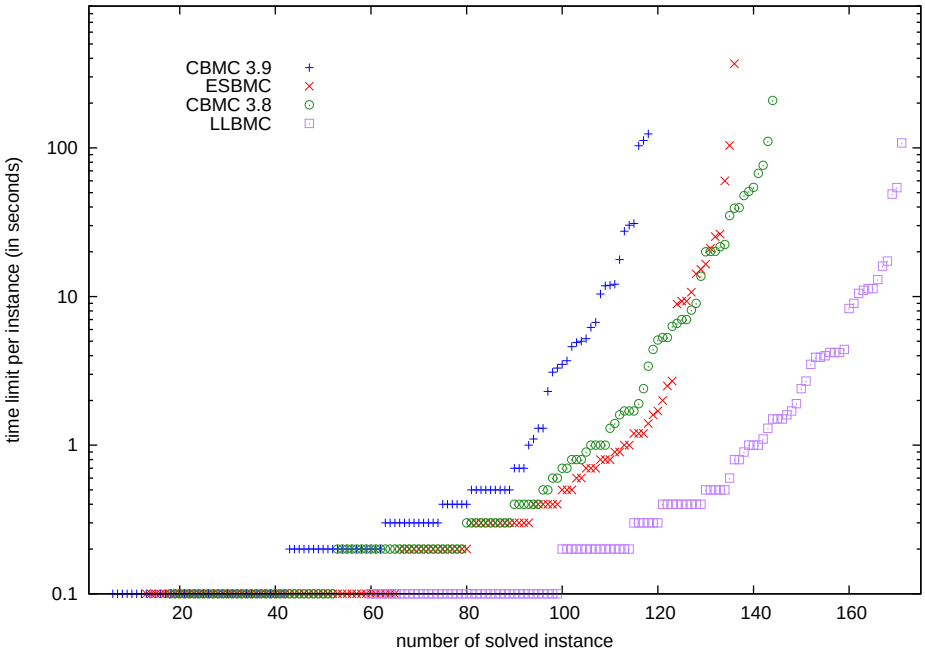


**Fig. 6.** Cactus plot comparing `LLBMC`, `CBMC` 3.8, `CBMC` 3.9, and `ESBMC`.

Also in 2004, NEC Laboratories America implemented a bounded model checking approach for C programs in the tool `F-Soft` as described in [15]. They differentiate their tool from `CBMC` mainly through a basic block-based approach instead of an SSA-based approach. Several static program analysis techniques are performed on the control-flow graph in order to simplify the BMC problem.

In 2009, Armando *et al.* extended `CBMC` to use SMT solvers instead of encoding the problem directly into SAT [1]. Results from that paper clearly show the benefits of using SMT solvers w.r.t. formula size and execution time compared to a direct SAT encoding as done by `CBMC` and `F-Soft`.

Recently, Cordeiro *et al.* presented `ESBMC` [10], which is based on `CBMC` but uses an SMT solver instead of a SAT solver. The main novelty of `ESBMC` is its added support for bug finding in multi-threaded software.

Milicevic and Kugler introduced an approach for model checking of software based on SMT and the theory of lists [21]. While that approach avoids the boundedness limitation of BMC, the evaluation in [21] indicates that it does not scale comparably to BMC based approaches.

*Symbolic execution* is a different approach to bug detection in programs. In contrast to BMC, which encodes all paths up to a bounded length in a single formula, symbolic execution performs a symbolic path exploration that considers the paths separately. The constraints obtained for each path are solved using SAT or SMT solvers. Recent symbolic execution tools include `KLEE` [7] for C programs

and KLOVER [19], which extends KLEE for C++ programs. Both KLEE and KLOVER perform symbolic execution on the level of LLVM-IR.

A recent tool that combines features of symbolic execution and BMC is LAV [27]. Like KLEE, KLOVER, and LLBMC, the tool LAV also operates on the level of LLVM-IR programs.

Out of the numerous static checking programs, at least Calysto [2] and SMACK [24] operate on the level of LLVM-IR as well.

For related work concerning memory models, we refer to [26, 12].

## 8 Conclusions and Future Work

This paper has presented LLBMC, a tool for bounded model checking of C/C++ programs. LLBMC uses the LLVM compiler framework to translate C/C++ programs into LLVM's intermediate representation. The resulting code is then converted into a logical representation and simplified using rewrite rules. The simplified formula is finally passed to an SMT solver. An empirical evaluation on a large collection of C programs has shown that LLBMC compares favorably to CBMC [8] and ESBMC [10], both in run-time and in number of found bugs. Furthermore, LLBMC has successfully been used on over 50 non-trivial C++ programs containing advanced features such as multiple inheritance, STL containers, and templates, making it (to the best of our knowledge) the first BMC tool that can handle non-trivial C++ programs.

For future work, we are currently working on lifting the error trace of the LLVM-IR program to an error trace of the C program by using debug information generated by the compiler front-end. We are also planning to determine (and iteratively adapt) loop unrolling and function inlining bounds automatically: starting with low bounds for function inlining and loop unrolling, they are gradually increased based on the results of previous runs of LLBMC. Since the C++ support in LLBMC is currently preliminary and incomplete, we are planning to extend the support for BMC of C++ programs. Similar to [19], support for exception handling and run-time type information (RTTI) needs to be added to LLBMC. Finally, LLBMC could be extended in the direction of software verification (as opposed to bug finding) using $k$-induction, similar to how this was recently done for CBMC [11].

## References


1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. STTT 11(1), 69–83 (2009)
2. Babić, D., Hu, A.J.: Calysto: Scalable and precise extended static checking. In: Proc. ICSE 2008. pp. 211–220 (2008)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS 1999. LNCS, vol. 1579, pp. 193–207 (1999)
4. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Proc. TACAS 2009. LNCS, vol. 5505, pp. 174–177 (2009)

5. Brummayer, R.D.: Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays. Ph.D. thesis, Johannes Kepler Universität, Linz, Austria (2009)
6. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: $10^{20}$ states and beyond. IC 98(2), 142–170 (1992)
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI 2008. pp. 209–224 (2008)
8. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS 2004. LNCS, vol. 2988, pp. 168–176 (2004)
9. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. ENTCS 254, 85–103 (2009)
10. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: Proc. ASE 2009. pp. 137–148 (2009)
11. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using $k$-induction. In: Proc. SAS 2011. LNCS, vol. 6887, pp. 351–368 (2011)
12. Falke, S., Merz, F., Sinz, C.: A theory of C-style memory allocation. In: Proc. SMT 2011. pp. 71–80 (2011)
13. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proc. CAV 2007. LNCS, vol. 4590, pp. 519–531 (2007)
14. Gustafsson, J., Betts, A., Ermedahl, A., Lisper, B.: The Mälardalen WCET benchmarks – past, present and future. In: Proc. WCET 2010. pp. 137–147 (2010)
15. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. TCS 404(3), 256–274 (2008)
16. Kim, M., Kim, Y., Kim, H.: Unit testing of flash memory device driver through a SAT-based model checker. In: Proc. ASE 2008. pp. 198–207 (2008)
17. Kröning, D.: CBMC release 3.9 announcement on `cprover@googlegroups.com` (December 19, 2010)
18. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO 2004. pp. 75–88 (2004)
19. Li, G., Ghosh, I., Rajan, S.: KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In: Proc. CAV 2011. pp. 609–615 (2011)
20. Maric, F., Janicic, P.: URBiVA: Uniform reduction to bit-vector arithmetic. In: Proc. IJCAR 2011. LNCS, vol. 6173, pp. 346–352 (2010)
21. Milicevic, A., Kugler, H.: Model checking using SMT and theory of lists. In: Proc. NFM 2011. LNCS, vol. 6617, pp. 282–297 (2011)
22. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS 2008. LNCS, vol. 4963, pp. 337–340 (2008)
23. Post, H., Sinz, C., Küchlin, W.: Towards automatic software model checking of thousands of Linux modules—A case study with Avinux. STVR 19(2), 155–172 (2009)
24. Rakamarić, Z., Hu, A.J.: A scalable memory model for low-level code. In: Proc. VMCAI 2009. LNCS, vol. 5403, pp. 290–304 (2009)
25. Sinha, N.: Symbolic program analysis using term rewriting and generalization. In: Proc. FMCAD 2008. pp. 1–9 (2008)
26. Sinz, C., Falke, S., Merz, F.: A precise memory model for low-level bounded model checking. In: Proc. SSV 2010 (2010)
27. Vujošević-Janičić, M., Kuncak, V.: Development and evaluation of LAV: An SMT-based error finding platform. In: Proc. VSTTE 2012. LNCS, vol. 7152, pp. 98–113 (2012)