Karlsruhe Institute of Technology

The Low-Level Bounded Model Checker LLBMC

Stephan Falke Florian Merz Carsten Sinz | May 27, 2010

VERIFICATION MEETS ALGORITHM ENGINEERING

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, …
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, ...
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

Karlsruhe Institute of Technology

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, ...
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

What is LLBMC?

- LLBMC = Low-Level (Software) Bounded Model Checking
  - Low-Level: Not operating on source code but on “abstract assembler”
  - Software: Programs written in C/C++/Objective C
  - Bounded: restricted number of nested function calls and loop iterations
  - Model Checking: “highly precise static analysis”
- Properties checked:
  - Built-in properties: invalid memory accesses, use after free, double free, range overflow, division by zero, . . .
  - User-supplied properties: `assert` statements
- Focus on memory properties

Motivation

The worldwide damage caused by malware (i.e. viruses, worms, Trojans) was $13.3 billion in 2006

Hacker attacks cost the world economy a whopping $1.6 trillion in 2000

Buffer overflows are still the number one issue as reported in operating system (OS) vendor advisories. (. . .) Integer overflows, barely in the top ten overall in the past few years, are number two for OS vendor advisories (in 2006), behind buffer overflows

Use-after-free vulnerability in Microsoft Internet Explorer (. . .) allows remote attackers to execute arbitrary code by accessing a pointer associated with a deleted object (. . .)

Motivation

The worldwide damage caused by malware (i.e. viruses, worms, Trojans) was $13.3 billion in 2006

Hacker attacks cost the world economy a whopping $1.6 trillion in 2000

Buffer overflows are still the number one issue as reported in operating system (OS) vendor advisories. (. . .) Integer overflows, barely in the top ten overall in the past few years, are number two for OS vendor advisories (in 2006), behind buffer overflows

Use-after-free vulnerability in Microsoft Internet Explorer (. . .) allows remote attackers to execute arbitrary code by accessing a pointer associated with a deleted object (. . .)

Motivation

The worldwide damage caused by malware (i.e. viruses, worms, Trojans) was $13.3 billion in 2006

Hacker attacks cost the world economy a whopping $1.6 trillion in 2000

Buffer overflows are still the number one issue as reported in operating system (OS) vendor advisories. (. . .) Integer overflows, barely in the top ten overall in the past few years, are number two for OS vendor advisories (in 2006), behind buffer overflows

Use-after-free vulnerability in Microsoft Internet Explorer (. . .) allows remote attackers to execute arbitrary code by accessing a pointer associated with a deleted object (. . .)

Motivation

The worldwide damage caused by malware (i.e. viruses, worms, Trojans) was $13.3 billion in 2006

Hacker attacks cost the world economy a whopping $1.6 trillion in 2000

Buffer overflows are still the number one issue as reported in operating system (OS) vendor advisories. (. . .) Integer overflows, barely in the top ten overall in the past few years, are number two for OS vendor advisories (in 2006), behind buffer overflows

Use-after-free vulnerability in Microsoft Internet Explorer (. . .) allows remote attackers to execute arbitrary code by accessing a pointer associated with a deleted object (. . .)

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Software Bounded Model Checking

- Programs deal with unbounded data structures such as linked lists, trees, etc.
- Property checking is thus undecidable
- Bugs manifest themselves in (typically short) finite runs of the program
- Software bounded model checking:
  - Analyze only bounded program runs
  - Restrict number of nested function calls and inline functions
  - Restrict number of loop iterations and unroll loops
  - Data structures are then bounded as well
  - Property checking becomes decidable by a logical encoding into SAT or SMT

Specifying and Verifying Properties

- Properties are typically formalized using `assume` and `assert` statements
  - `assume` states a pre-condition that is assumed to hold at its location
  - `assert` states a post-condition that is to be checked at its location
- The program `Prog` is correct if

$$\texttt{Prog} \wedge \bigwedge \texttt{assume} \Rightarrow \bigwedge \texttt{assert}$$

  is valid
- In software bounded model checking, this can be decided using a SAT or SMT solver

Specifying and Verifying Properties

- Properties are typically formalized using `assume` and `assert` statements
  - `assume` states a pre-condition that is assumed to hold at its location
  - `assert` states a post-condition that is to be checked at its location
- The program `Prog` is correct if

  $$\texttt{Prog} \wedge \bigwedge \texttt{assume} \Rightarrow \bigwedge \texttt{assert}$$

  is valid
- In software bounded model checking, this can be decided using a SAT or SMT solver

Specifying and Verifying Properties

- Properties are typically formalized using `assume` and `assert` statements
  - `assume` states a pre-condition that is assumed to hold at its location
  - `assert` states a post-condition that is to be checked at its location
- The program `Prog` is correct if

$$\mathtt{Prog} \wedge \bigwedge \mathtt{assume} \Rightarrow \bigwedge \mathtt{assert}$$

  is valid
- In software bounded model checking, this can be decided using a SAT or SMT solver

Specifying and Verifying Properties

- Properties are typically formalized using `assume` and `assert` statements
  - `assume` states a pre-condition that is assumed to hold at its location
  - `assert` states a post-condition that is to be checked at its location
- The program `Prog` is correct if

$$\texttt{Prog} \wedge \bigwedge \texttt{assume} \Rightarrow \bigwedge \texttt{assert}$$

  is valid
- In software bounded model checking, this can be decided using a SAT or SMT solver

Specifying and Verifying Properties

- Properties are typically formalized using `assume` and `assert` statements
  - `assume` states a pre-condition that is assumed to hold at its location
  - `assert` states a post-condition that is to be checked at its location
- The program `Prog` is correct if

$$\mathtt{Prog} \wedge \bigwedge \mathtt{assume} \Rightarrow \bigwedge \mathtt{assert}$$

  is valid
- In software bounded model checking, this can be decided using a SAT or SMT solver

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
  - Closer to the code that is actually run
  - Well-defined, simple semantics makes logical encoding easier
  - Compiler optimizations etc. come "for free"
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
  - Closer to the code that is actually run
  - Well-defined, simple semantics makes logical encoding easier
  - Compiler optimizations etc. come “for free”
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
    - Closer to the code that is actually run
    - Well-defined, simple semantics makes logical encoding easier
    - Compiler optimizations etc. come “for free”
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
  - Closer to the code that is actually run
  - Well-defined, simple semantics makes logical encoding easier
  - Compiler optimizations etc. come "for free"
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
  - Closer to the code that is actually run
  - Well-defined, simple semantics makes logical encoding easier
  - Compiler optimizations etc. come “for free”
- LLBMC uses the LLVM intermediate language and compiler infrastructure

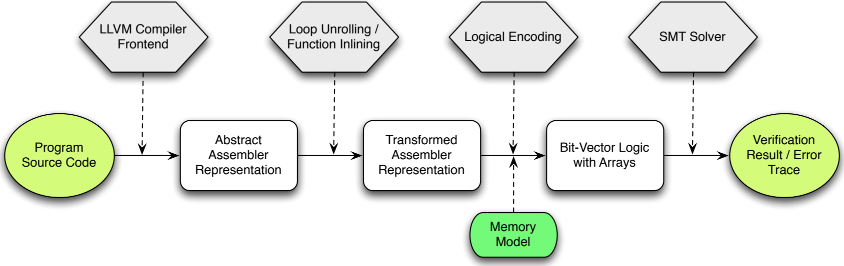
Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
    - Closer to the code that is actually run
    - Well-defined, simple semantics makes logical encoding easier
    - Compiler optimizations etc. come “for free”
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Low Level Bounded Model Checking

- Fully supporting real-life programming languages is cumbersome
- Particularly true for C/C++/Objective C due to their complex (sometimes ambiguous) semantics
- Key idea: Instead of operating on the source code directly it is easier to operate on a compiler intermediate language (abstract assembler)
  - Closer to the code that is actually run
  - Well-defined, simple semantics makes logical encoding easier
  - Compiler optimizations etc. come “for free”
- LLBMC uses the LLVM intermediate language and compiler infrastructure

Overview of the LLBMC Approach


LLVM Compiler Frontend

Loop Unrolling / Function Inlining

Logical Encoding

SMT Solver

Program Source Code

Abstract Assembler Representation

Transformed Assembler Representation

Bit-Vector Logic with Arrays

Verification Result / Error Trace

Memory Model

Memory model captures the semantics of memory accesses

Example

```c
struct S {
    int x;
    struct S *n;
};

int main(int argc, char *argv[]) {
    struct S *p, *q;

    p = malloc(sizeof(struct S));
    p->x = 5;
    p->n = NULL;

    if (argc > 1) {
        q = malloc(sizeof(struct S));
        q->x = 5;
        q->n = p;
    } else {
        q = p;
    }

    __llbmc_assert(p->x + q->x == 10);

    free(q);
    free(p);

    return 0;
}
```

```llvm
%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*
  %p.x = getelementptr %struct.S* %p, i32 0, i32 0
  store i32 5, i32* %p.x
  %p.n = getelementptr %struct.S* %p, i32 0, i32 1
  store %struct.S* null, %struct.S** %p.n
  %c.1 = icmp sgt i32 %argc, 1
  br i1 %c.1, label %if.then, label %if.end

if.then:
  %1 = call i8* @malloc(i32 8)
  %q = bitcast i8* %1 to %struct.S*
  %q.x = getelementptr %struct.S* %q, i32 0, i32 0
  store i32 5, i32* %q.x
  %q.n = getelementptr %struct.S* %q, i32 0, i32 1
  store %struct.S* %p, %struct.S** %q.n
  br label %if.end

if.end:
  %q.0 = phi %struct.S* [ %q, %if.then ], [ %p, %entry ]
  %q.0.x = getelementptr %struct.S* %q.0, i32 0, i32 0
  %2 = load i32* %p.x
  %3 = load i32* %q.0.x
  %4 = add i32 %2, %3
  %c.2 = icmp eq i32 %4, 10
  %5 = zext i1 %c.2 to i32
  call void @__llbmc_assert(i32 %5)
  %6 = bitcast %struct.S* %q.0 to i8*
  call void @free(i8* %6)
  %7 = bitcast %struct.S* %p to i8*
  call void @free(i8* %7)
  ret i32 0
}
```

Example

```c
struct S {
    int x;
    struct S *n;
};

int main(int argc, char *argv[]) {
    struct S *p, *q;

    p = malloc(sizeof(struct S));
    p->x = 5;
    p->n = NULL;

    if (argc > 1) {
        q = malloc(sizeof(struct S));
        q->x = 5;
        q->n = p;
    } else {
        q = p;
    }

    __llbmc_assert(p->x + q->x == 10);

    free(q);
    free(p);

    return 0;
}
```

```llvm
%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*
  %p.x = getelementptr %struct.S* %p, i32 0, i32 0
  store i32 5, i32* %p.x
  %p.n = getelementptr %struct.S* %p, i32 0, i32 1
  store %struct.S* null, %struct.S** %p.n
  %c.1 = icmp sgt i32 %argc, 1
  br i1 %c.1, label %if.then, label %if.end

if.then:
  %1 = call i8* @malloc(i32 8)
  %q = bitcast i8* %1 to %struct.S*
  %q.x = getelementptr %struct.S* %q, i32 0, i32 0
  store i32 5, i32* %q.x
  %q.n = getelementptr %struct.S* %q, i32 0, i32 1
  store %struct.S* %p, %struct.S** %q.n
  br label %if.end

if.end:
  %q.0 = phi %struct.S* [ %q, %if.then ], [ %p, %entry ]
  %q.0.x = getelementptr %struct.S* %q.0, i32 0, i32 0
  %2 = load i32* %p.x
  %3 = load i32* %q.0.x
  %4 = add i32 %2, %3
  %c.2 = icmp eq i32 %4, 10
  %5 = zext i1 %c.2 to i32
  call void @__llbmc_assert(i32 %5)
  %6 = bitcast %struct.S* %q.0 to i8*
  call void @free(i8* %6)
  %7 = bitcast %struct.S* %p to i8*
  call void @free(i8* %7)
  ret i32 0
}
```

Encoding of `phi`-Instructions

- The abstract assembler contains `phi`-instructions of the form

$$i' = \texttt{phi}[i_1, bb_1], \ldots, [i_n, bb_n]$$

where $bb_1, \ldots, bb_n$ are basic blocks

- For the logical encoding, bb_j is replaced by

$$c_{\texttt{exec}}(bb_j) \land t(bb_j, b)$$

where

  - b is the basic block containing the `phi`-instruction
  - $c_{\texttt{exec}}(bb_j)$ is bb_j's execution condition
  - $t(bb_j, b)$ is the condition under which control passes from bb_j to b

Encoding of `phi`-Instructions

- The abstract assembler contains `phi`-instructions of the form

$$i' = \texttt{phi}[i_1, bb_1], \ldots, [i_n, bb_n]$$

  where $bb_1, \ldots, bb_n$ are basic blocks

- For the logical encoding, bb_j is replaced by

$$c_{\texttt{exec}}(bb_j) \wedge t(bb_j, b)$$

  where

  - b is the basic block containing the `phi`-instruction
  - $c_{\texttt{exec}}(bb_j)$ is bb_j's execution condition
  - $t(bb_j, b)$ is the condition under which control passes from bb_j to b

Elimination of branches

- The memory can be modelled as an array of bytes
- Bring the memory into SSA form by introducing an abstract type `memstate`
  - Memory is accessed using `read`-instructions
  - Memory is changed using `write`-, `malloc`-, and `free`-instructions
  - `phi`-instructions for memory states are introduced
- With the encoding of `phi`-instructions and the conversion of the memory in SSA form branches can be eliminated

Elimination of branches

- The memory can be modelled as an array of bytes
- Bring the memory into SSA form by introducing an abstract type `memstate`
  - Memory is accessed using `read`-instructions
  - Memory is changed using `write`-, `malloc`-, and `free`-instructions
  - `phi`-instructions for memory states are introduced
- With the encoding of `phi`-instructions and the conversion of the memory in SSA form branches can be eliminated

Elimination of branches

- The memory can be modelled as an array of bytes
- Bring the memory into SSA form by introducing an abstract type `memstate`
  - Memory is accessed using `read`-instructions
  - Memory is changed using `write`-, `malloc`-, and `free`-instructions
  - `phi`-instructions for memory states are introduced
- With the encoding of `phi`-instructions and the conversion of the memory in SSA form branches can be eliminated

Example

```
%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*
  %p.x = getelementptr %struct.S* %p, i32 0, i32 0
  store i32 5, i32* %p.x
  %p.n = getelementptr %struct.S* %p, i32 0, i32 1
  store %struct.S* null, %struct.S** %p.n
  %c.1 = icmp sgt i32 %argc, 1
  br i1 %c.1, label %if.then, label %if.end

if.then:
  %1 = call i8* @malloc(i32 8)
  %q = bitcast i8* %1 to %struct.S*
  %q.x = getelementptr %struct.S* %q, i32 0, i32 0
  store i32 5, i32* %q.x
  %q.n = getelementptr %struct.S* %q, i32 0, i32 1
  store %struct.S* %p, %struct.S** %q.n
  br label %if.end

if.end:
  %q.0 = phi %struct.S* [ %q, %if.then ], [ %p, %entry ]
  %q.0.x = getelementptr %struct.S* %q.0, i32 0, i32 0
  %2 = load i32* %p.x
  %3 = load i32* %q.0.x
  %4 = add i32 %2, %3
  %c.2 = icmp eq i32 %4, 10
  %5 = zext i1 %c.2 to i32
  call void @__llbmc_assert(i32 %5)
  %6 = bitcast %struct.S* %q.0 to i8*
  call void @free(i8* %6)
  %7 = bitcast %struct.S* %p to i8*
  call void @free(i8* %7)
  ret i32 0
}
```

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

memstate %mem0
i8* %0
memstate %mem1 = malloc(%mem0, %0, 8)
struct.S* %p = bitcast(%0)
i32* %p.x = getelementptr(%p, 0, 0)
memstate %mem2 = store(%mem1, %p.x, 5)
struct.S** %p.n = getelementptr(%p, 0, 1)
memstate %mem3 = store(%mem2, %p.n, null)
i32 %argc
i1 %c.1 = %argc > 1

i8* %1
memstate %mem4 = malloc(%mem3, %1, 8)
struct.S* %q = bitcast(%1)
i32* %q.x = getelementptr(%q, 0, 0)
memstate %mem5 = store(%mem4, %q.x, 5)
struct.S** %q.n = getelementptr(%q, 0, 1)
memstate %mem6 = store(%mem5, %q.n, %p)

memstate %mem7 = phi([%mem3, !%c.1], [%mem6, %c.1])
struct.S* %q.0 = phi([%p, !%c.1], [%q, %c.1])
i32* %q.0.x = getelementptr(%q.0, 0, 0)
i32 %2 = load(%mem7, %p.x)
i32 %3 = load(%mem7, %q.0.x)
i32 %4 = add(%2, %3)
i1 %c.2 = %4 == 10
assert(%c.2)
memstate %mem8 = free(%mem7, %q.0)
memstate %mem9 = free(%mem8, %p);
```

Example

```
%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*
  %p.x = getelementptr %struct.S* %p, i32 0, i32 0
  store i32 5, i32* %p.x
  %p.n = getelementptr %struct.S* %p, i32 0, i32 1
  store %struct.S* null, %struct.S** %p.n
  %c.1 = icmp sgt i32 %argc, 1
  br i1 %c.1, label %if.then, label %if.end

if.then:
  %1 = call i8* @malloc(i32 8)
  %q = bitcast i8* %1 to %struct.S*
  %q.x = getelementptr %struct.S* %q, i32 0, i32 0
  store i32 5, i32* %q.x
  %q.n = getelementptr %struct.S* %q, i32 0, i32 1
  store %struct.S* %p, %struct.S** %q.n
  br label %if.end

if.end:
  %q.0 = phi %struct.S* [ %q, %if.then ], [ %p, %entry ]
  %q.0.x = getelementptr %struct.S* %q.0, i32 0, i32 0
  %2 = load i32* %p.x
  %3 = load i32* %q.0.x
  %4 = add i32 %2, %3
  %c.2 = icmp eq i32 %4, 10
  %5 = zext i1 %c.2 to i32
  call void @__llbmc_assert(i32 %5)
  %6 = bitcast %struct.S* %q.0 to i8*
  call void @free(i8* %6)
  %7 = bitcast %struct.S* %p to i8*
  call void @free(i8* %7)
  ret i32 0
}
```

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

memstate %mem0
i8* %0
memstate %mem1 = malloc(%mem0, %0, 8)
struct.S* %p = bitcast(%0)
i32* %p.x = getelementptr(%p, 0, 0)
memstate %mem2 = store(%mem1, %p.x, 5)
struct.S** %p.n = getelementptr(%p, 0, 1)
memstate %mem3 = store(%mem2, %p.n, null)
i32 %argc
i1 %c.1 = %argc > 1

i8* %1
memstate %mem4 = malloc(%mem3, %1, 8)
struct.S* %q = bitcast(%1)
i32* %q.x = getelementptr(%q, 0, 0)
memstate %mem5 = store(%mem4, %q.x, 5)
struct.S** %q.n = getelementptr(%q, 0, 1)
memstate %mem6 = store(%mem5, %q.n, %p)

memstate %mem7 = phi([%mem3, !%c.1], [%mem6, %c.1])
struct.S* %q.0 = phi([%p, !%c.1], [%q, %c.1])
i32* %q.0.x = getelementptr(%q.0, 0, 0)
i32 %2 = load(%mem7, %p.x)
i32 %3 = load(%mem7, %q.0.x)
i32 %4 = add(%2, %3)
i1 %c.2 = %4 == 10
assert(%c.2)
memstate %mem8 = free(%mem7, %q.0)
memstate %mem9 = free(%mem8, %p);
```

Memory Modification Graph

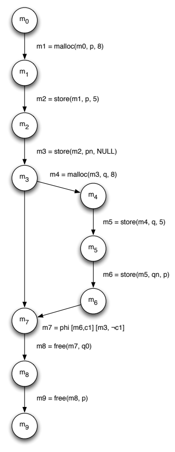
Example

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

memstate %mem0
i8* %0
memstate %mem1 = malloc(%mem0, %0, 8)
struct.S* %p = bitcast(%0)
i32* %p.x = getelementptr(%p, 0, 0)
memstate %mem2 = store(%mem1, %p.x, 5)
struct.S** %p.n = getelementptr(%p, 0, 1)
memstate %mem3 = store(%mem2, %p.n, null)
i32 %argc
i1 %c.1 = %argc > 1

i8* %1
memstate %mem4 = malloc(%mem3, %1, 8)
struct.S* %q = bitcast(%1)
i32* %q.x = getelementptr(%q, 0, 0)
memstate %mem5 = store(%mem4, %q.x, 5)
struct.S** %q.n = getelementptr(%q, 0, 1)
memstate %mem6 = store(%mem5, %q.n, %p)

memstate %mem7 = phi([%mem3, !%c.1], [%mem6, %c.1])
struct.S* %q.0 = phi([%p, !%c.1], [%q, %c.1])
i32* %q.0.x = getelementptr(%q.0, 0, 0)
i32 %2 = load(%mem7, %p.x)
i32 %3 = load(%mem7, %q.0.x)
i32 %4 = add(%2, %3)
i1 %c.2 = %4 == 10
assert(%c.2)
memstate %mem8 = free(%mem7, %q.0)
memstate %mem9 = free(%mem8, %p);
```



Encoding Memory Constraints 1

- The following memory checks are built-in:
  - Valid read/writes (i.e., only to allocated memory)
  - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
  - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
  - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
  - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
  - $(\ldots)$

Encoding Memory Constraints 1

- The following memory checks are built-in:
  - Valid read/writes (i.e., only to allocated memory)
  - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
  - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
  - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
  - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
  - $(\ldots)$

Encoding Memory Constraints 1

- The following memory checks are built-in:
    - Valid read/writes (i.e., only to allocated memory)
    - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
    - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
    - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
    - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
    - $(\ldots)$

Encoding Memory Constraints 1

- The following memory checks are built-in:
  - Valid read/writes (i.e., only to allocated memory)
  - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
  - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
  - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
  - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
  - $(\ldots)$

Encoding Memory Constraints 1

- The following memory checks are built-in:
  - Valid read/writes (i.e., only to allocated memory)
  - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
  - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
  - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
  - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
  - $(\ldots)$

Encoding Memory Constraints 1

- The following memory checks are built-in:
  - Valid read/writes (i.e., only to allocated memory)
  - Valid frees (i.e., `free` is only called for the beginning of a block of allocated memory)
  - No double frees (i.e., no memory block is `free`'d twice)
- Building blocks:
  - `valid_mem_access`(m, p, s): the range $p, \ldots, p + s - 1$ is allocated in m
  - `deallocated`(m, m', p): the block beginning at p is `free`'d between m and m'
  - $(\ldots)$

Encoding Memory Constraints 2

$m \preceq m'$: there exists a path from m to m' in the memory modification graph

$c_{\texttt{exec}}(I)$: execution condition of the (basic block containing the) instruction I

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

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

Encoding Memory Constraints 2

$m \preceq m'$: there exists a path from m to m' in the memory modification graph

$c_{\texttt{exec}}(I)$: execution condition of the (basic block containing the) instruction I

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

$$\texttt{valid\_mem\_access}(m, p, s) \;\equiv\; \bigvee_{\substack{m' \preceq m \\ I\colon\, m' = \texttt{malloc}(\hat{m}, q, t)}} \begin{array}{l} c_{\texttt{exec}}(I) \;\wedge (q \leq p \leq q + t - s) \\ \qquad\quad \wedge \neg\texttt{deallocated}(m', m, q) \end{array}$$

Encoding Memory Constraints 3

- Each $m' = \texttt{write}(m, p, x)$ and each $x = \texttt{read}(m, p)$ is preceeded by the assertion

  $$\texttt{valid\_mem\_access}(m, p, s)$$

  where s is the appropriate size
- Similar assertions are added for the other built-in memory checks
- For `malloc`-instructions, assumptions on disjointness of the allocated memory regions are added

Encoding Memory Constraints 3

- Each $m' = \texttt{write}(m, p, x)$ and each $x = \texttt{read}(m, p)$ is preceeded by the assertion

$$\texttt{valid\_mem\_access}(m, p, s)$$

  where s is the appropriate size
- Similar assertions are added for the other built-in memory checks
- For `malloc`-instructions, assumptions on disjointness of the allocated memory regions are added

Encoding Memory Constraints 3

- Each $m' = \texttt{write}(m, p, x)$ and each $x = \texttt{read}(m, p)$ is preceeded by the assertion

  $$\texttt{valid\_mem\_access}(m, p, s)$$

  where s is the appropriate size
- Similar assertions are added for the other built-in memory checks
- For `malloc`-instructions, assumptions on disjointness of the allocated memory regions are added

Example

```
struct.S = struct { i32, struct.S* }
memstate %initialMemState
i8* %0
i1 %2 = 0x00000000 <= (void*)%0
i32 %4 = add((i32)%0, 7)
i1 %6 = 0x5fffffff >= (void*)%4
i1 %7 = (void*)%0 <= (void*)%4
i1 %8 = and(%2, %6)
i1 %9 = and(%8, %7)
assume(malloc_assume, %9, 1)
memstate %11 = malloc(heap, %initialMemState, %0, 8, 1)
i32* %p.x = getelementptr((struct.S*)%0, 0, 0)
i1 %13 = 0xbfffffff < (void*)%p.x
i32 %14 = add((i32)%p.x, 3)
i1 %16 = 0xbfffffff >= (void*)%14
i1 %17 = and(%13, %16)
i1 %18 = %0 <= %p.x
i32 %19 = add((i32)%p.x, 4)
i32 %21 = add((i32)%0, 8)
i1 %23 = (void*)%19 <= (void*)%21
i1 %24 = and(%18, %23)
i1 %25 = or(%17, %24)
assert(valid_store, %25, 1)
memstate %27 = store(%11, %p.x, 5, 1)
struct.S** %p.n = getelementptr((struct.S*)%0, 0, 1)
i1 %29 = 0xbfffffff < (void*)%p.n
i32 %30 = add((i32)%p.n, 3)
i1 %32 = 0xbfffffff >= (void*)%30
i1 %33 = and(%29, %32)
i1 %34 = %0 <= %p.n
i32 %35 = add((i32)%p.n, 4)
i1 %37 = (void*)%35 <= (void*)%21
i1 %38 = and(%34, %37)
i1 %39 = or(%33, %38)
assert(valid_store, %39, 1)
memstate %41 = store(%27, %p.n, 0x00000000, 1)
i32 %argc
i1 %c.1 = %argc > 1
i8* %42
i1 %44 = 0x00000000 <= (void*)%42
i32 %46 = add((i32)%42, 7)
i1 %48 = 0x5fffffff >= (void*)%46
i1 %49 = (void*)%42 <= (void*)%46
i1 %50 = and(%44, %48)
i1 %51 = and(%50, %49)
i32 %52 = add((i32)%42, 8)
i1 %54 = (void*)%52 <= (void*)%0
i1 %55 = (void*)%21 <= (void*)%42
i1 %56 = or(%54, %55)
i1 %57 = and(%51, %56)
assume(malloc_assume, %57, %c.1)
memstate %59 = malloc(heap, %41, %42, 8, %c.1)
i32* %q.x = getelementptr((struct.S*)%42, 0, 0)
i1 %61 = 0xbfffffff < (void*)%q.x
i32 %62 = add((i32)%q.x, 3)
i1 %64 = 0xbfffffff >= (void*)%62
i1 %65 = and(%61, %64)
i1 %66 = %0 <= %q.x
i32 %67 = add((i32)%q.x, 4)
i1 %69 = (void*)%67 <= (void*)%21
i1 %70 = and(%66, %69)
i1 %71 = %42 <= %q.x
i1 %72 = (void*)%67 <= (void*)%52
i1 %73 = and(%71, %72)
i1 %74 = and(%c.1, %73)
i1 %75 = or(%70, %74)
i1 %76 = or(%65, %75)
assert(valid_store, %76, %c.1)
```

```
assert(valid_store, %76, %c.1)
memstate %78 = store(%59, %q.x, 5, %c.1)
struct.S** %q.n = getelementptr((struct.S*)%42, 0, 1)
i1 %80 = 0xbfffffff < (void*)%q.n
i32 %81 = add((i32)%q.n, 3)
i1 %83 = 0xbfffffff >= (void*)%81
i1 %84 = and(%80, %83)
i1 %85 = %0 <= %q.n
i32 %86 = add((i32)%q.n, 4)
i1 %88 = (void*)%86 <= (void*)%21
i1 %89 = and(%85, %88)
i1 %90 = %42 <= %q.n
i1 %91 = (void*)%86 <= (void*)%52
i1 %92 = and(%90, %91)
i1 %93 = and(%c.1, %92)
i1 %94 = or(%89, %93)
i1 %95 = or(%84, %94)
assert(valid_store, %95, %c.1)
memstate %97 = store(%78, %q.n, (struct.S*)%0, %c.1)
void* %stacktopptr0 = phi([0xbfffffff, !%c.1], [0xbfffffff, %c.1])
memstate %if.end__mem = phi([%41, !%c.1], [%97, %c.1])
struct.S* %q.0 = phi([(struct.S*)%0, !%c.1], [(struct.S*)%42, %c.1])
i32* %q.0.x = getelementptr(%q.0, 0, 0)
i1 %98 = %stacktopptr0 < (void*)%p.x
i1 %99 = and(%98, %16)
i1 %100 = %42 <= %p.x
i1 %101 = (void*)%19 <= (void*)%52
i1 %102 = and(%100, %101)
i1 %103 = and(%c.1, %102)
i1 %104 = or(%24, %103)
i1 %105 = or(%99, %104)
assert(valid_load, %105, 1)
i32 %107 = load(%if.end__mem, %p.x, 1)
i1 %109 = %stacktopptr0 < (void*)%q.0.x
i32 %110 = add((i32)%q.0.x, 3)
i1 %112 = 0xbfffffff >= (void*)%110
i1 %113 = and(%109, %112)
i1 %114 = %0 <= %q.0.x
i32 %115 = add((i32)%q.0.x, 4)
i1 %117 = (void*)%115 <= (void*)%21
i1 %118 = and(%114, %117)
i1 %119 = %42 <= %q.0.x
i1 %120 = (void*)%115 <= (void*)%52
i1 %121 = and(%119, %120)
i1 %122 = and(%c.1, %121)
i1 %123 = or(%118, %122)
i1 %124 = or(%113, %123)
assert(valid_load, %124, 1)
i32 %126 = load(%if.end__mem, %q.0.x, 1)
i32 %127 = add(%107, %126)
i1 %c.2 = %127 == 10
i1 %129 = (i8*)%q.0 == %0
i1 %130 = (i8*)%q.0 == %42
i1 %131 = and(%c.1, %130)
i1 %132 = or(%129, %131)
assert(valid_free, %132, %c.2)
i1 %134 = %0 == %0
i1 %135 = %0 == (i8*)%q.0
i1 %136 = and(%c.2, %135)
i1 %138 = and(%134, !%136)
i1 %139 = %0 == %42
i1 %140 = %42 == (i8*)%q.0
i1 %141 = and(%c.2, %140)
i1 %143 = and(%139, !%141)
i1 %144 = and(%c.1, %143)
i1 %145 = or(%138, %144)
assert(valid_free, %145, %c.2)
assert(custom, 0, !%c.2)
```

Example (Memory Management)

```
struct S {
    int x;
    struct S *n;
};

int main(int argc, char *argv[]) {
    struct S *p, *q;

    p = malloc(sizeof(struct S));
    p->x = 5;
    p->n = NULL;

    if (argc > 1) {
        q = malloc(sizeof(struct S));
        q->x = 5;
        q->n = p;
    } else {
        q = p;
    }

    __llbmc_assert(p->x + q->x == 10);

    free(q);
    free(p);

    return 0;
}
```

Example (Functional Correctness)

```c
int npo2(int x) {
    unsigned int i;
    x--;
    for(i = 1; i < sizeof(int) * 8; i *= 2) {
        x = x | (x >> i);
    }
    return x + 1;
}

int main(int argc, char *argv[]) {
    int x = argc;

    __llbmc_assume(x > 0 && x < (INT_MAX >> 1));

    int n = npo2(x);

    __llbmc_assert(n >= x);
    __llbmc_assert(n < (x << 1));
    __llbmc_assert((n & (n - 1)) == 0);

    return 0;
}
```

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
  - Rewriting
  - Restricted linear arithmetic
  - Boolean simplification
  - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)

Future Work

- Optimization of memory constraints
- Discharging of simple memory constraints using
    - Rewriting
    - Restricted linear arithmetic
    - Boolean simplification
    - (...)
- Iterative deepening of function inlining/loop unrolling
- Modular verification
- Handling of system calls (strings, memory copy, etc.)