Checker: a Static Program Checker

Nicholas Lewycky

June 7, 2006

Abstract

Automated software analysis is the process of testing program source code against a set of conditions. These may be as simple as verifying the coding standards, or as complicated as new languages which are formally verifiable by a theorem solver.

Checker is able to find two small classes of errors, one is memory faults, the other, non-deterministic behaviour. Lacking interprocedural analysis, checker can not be applied to real-world software.

Contents

1 Introduction

The first stop for checking a program for errors is the compiler. Modern compiler design focusses on efficient algorithms for fast compilation, with compiler warnings coming as a convenience.

I examine the possibility of using modern compiler techniques to produce a program checker. This is built on top of Chris Lattner's excellent compiler framework, LLVM (Low Level Virtual Machine) which provides a C front end, an SSA (Static Single Assignment) based intermediate representation and many different analysis and optimization passes.

1.1 Static Single Assignment

In source code form, a program declares variables and may assign and reassign values to them throughout the execution. In static single assignment form, there is one unchanging definition per variable. The conversion is done by defining a new variable at each assignment. This transforms `x = 1; y = x; x = 2;` into $x_1 \leftarrow 1; y_1 \leftarrow x_1; x_2 \leftarrow 2$.

A variable may be defined as being equal to multiple values by employing the ϕ function. Setting a value to $\phi(x_1, ..., x_n)$ means that the value is equal to one of the values in the set. Consider this simple example:

```
int x, y;
if (condition)
  x = a;
else
  x = b;
y = x;
```

In each branch, we have a different definition of the variable, $x_1 \leftarrow a$ and $x_2 \leftarrow b$. y is defined with the ϕ function. In our example above, $y \leftarrow \phi(x_1, x_2)$.

Similarly, loops are written with ϕ-nodes at the top. A loop with iterator i from 0 is written $i_1 \leftarrow \phi(0, i_1 + 1)$. An example:

```
int y = 0;
for (int i = 0; i < n; i++)
  y += i;
```

becomes:

```
$y_1 \leftarrow 0$;
for ($i_1 \leftarrow \phi(0, i_2)$; $i_1 < n$; $i_2 \leftarrow i_1 + 1$)
  $y_2 \leftarrow \phi(y_1, y_2 + i_1)$
```

LLVM uses static single assignment form for its intermediate representation. In LLVM's implementation of ϕ-nodes, each value in the set is associated with the basic block containing the definition of the variable.

1.2 Basic Blocks

A basic block is a series of instructions that execute in order with other points of entry or exit. In this sense it is atomic; execution will always begin at the beginning of a basic block and continue to the last instruction. Sometimes this rule is weakened to allow for error handling (such as division by zero) or exceptions.

One way to build basic blocks is to take the code and cut it wherever an instruction jumps, or wherever a jump may land. These form the boundaries of your basic blocks. Note that this technique doesn't always produce the largest possible basic blocks (and thus the fewest number).

In LLVM, a basic block must begin with the ϕ-nodes and end with a terminator instruction. These include branch instructions, calls of functions that may throw an exception, or a function return. Function calls that can not throw exceptions are not terminator instructions.

1.3 Control Flow Graph

Basic blocks form the nodes of a directed graph called the control flow graph. The edges connect one basic block to the blocks it could jump into. In this form, a cycle represents a loop, a detached subgraph or basic block is unreachable code, and a cycle with no edges leading out of it must be an infinite loop.

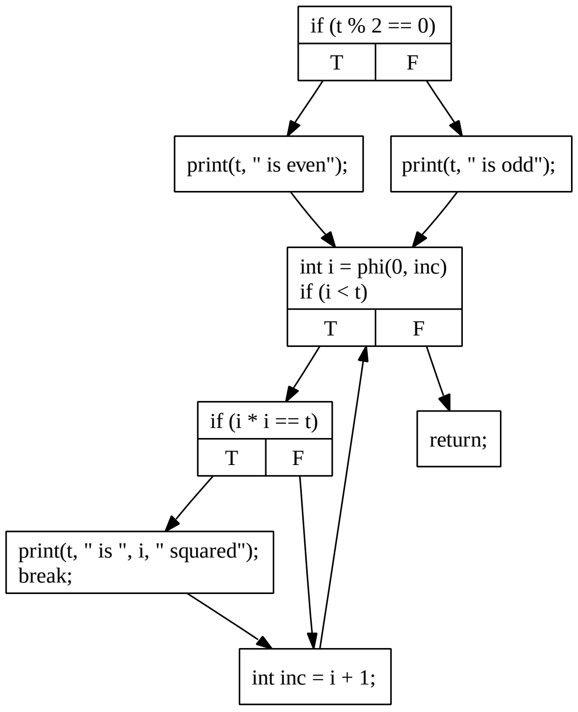
The program

```
if (t % 2 == 0)
  print(t, " is even");
else
  print(t, " is odd");

for (int i = 0; i < t; ++i)
  if (i * i == t) {
    print(t, " is ", i, " squared");
    break;
  }

return;
```

could be translated into:



1.4 LLVM Intermediate Representation

LLVM sports an instruction set architecture with explicit basic blocks, SSA form and 34 instructions in six categories. It features an explicit control flow

Type	Description	Type	Description
void	No value		
bool	True or Value		
ubyte	Unsigned byte	sbyte	Signed byte
ushort	Unsigned 16 bits	short	Signed 16 bits
uint	Unsigned 32 bits	int	Signed 32 bits
ulong	Unsigned 64 bits	long	Signed 64 bits
float	32-bit floating point	double	64-bit floating point
label	Branch destination		

Table 1: Primitive types in LLVM IR

graph, language-independent type information and explicit typed pointer arithmetic. It has an infinite number of registers and explicit memory operations.[ALB+03, LA04b]

There are 13 primitive types shown in Table 1 from which compound types may be derived. These derived types may be arrays, such as `[3 x [4 x int]]`, a pointer type `float*`, a structure {`int, float, [3 x sbyte]`} or a function `void (int)*` being a pointer to a function that returns void and takes a signed 32-bit integer.[Lat06]

The registers in LLVM are SSA variables, and their identifiers are prefixed with "%". Variables are declared by the statement that defines them. All instructions in LLVM are strongly typed, with the type of its arguments appearing before the argument. In the event that the two arguments of an binary operation are the same, the type is only specified once, immediately after the opcode. A sample instruction is `%result = mul uint %X, 8`.

Throughout this paper, I will refer to certain instructions:

load `%y = load` *`type*`* `%x`.
Dereferences the pointer `x` and loads the value into register `y`.

store `store` *`type`* `%x,` *`type*`* `%y`.
Stores the value in register `x` into the memory referred to by `y`. In C, this would be `*y = x;`.

getelementptr `%y = getelementptr` *`type*`* `%x, long 0, long 0`
"getelementptr" may be used as an instruction defining a variable, or inline as an expression in a another operation. In either case, "getelementptr" is variadic, meaning that is may have any number of argu-

ments. The first argument is always a base pointer, followed by the indices in the new pointer being calculated.

This is an especially interesting instruction because it allows the type-safe subset of pointer arithmetic.

malloc, alloca `%x = alloca` *`type`*
`%x = malloc` *`type`*
Both heap and stack allocation (malloc and alloca respectively) have their own opcodes, and take the same arguments: one type, optionally followed by a count, optionally followed by an alignment.

free `free` *`type`* `%x`
Frees memory allocated with the `malloc` instruction.

br `br label %loopexit`
`br bool %condition label %else label %then`
A branch statement, with an optional conditional.

Taken directly from LLVM's C front end, a small complete program looks like:

```
%y = global sbyte* %x           ; <sbyte**> [#uses=1]
%x = weak global sbyte 0                ; <sbyte*> [#uses=1]
%.str_1 = internal constant [6 x sbyte] c"abcde\00"             ; <[6 x sbyte]*>

implementation   ; Functions:

void %f() {
entry:
        %tmp.0 = load sbyte** %y                ; <sbyte*> [#uses=1]
        store sbyte 100, sbyte* %tmp.0
        ret void
}
```

The global variables begin a program, followed by the word "implementation", then the functions. Note that the label "entry" is mandatory; there is a label at the beginning of every basic block. The `; text` indicates a comment running to the end of the line. For the most part, the LLVM assembly language is self-explanatory, but for comprehensive details, please consult the language reference.[Lat06].

2 Related Work

The design of the checker was heavily influenced by two projects, Valgrind[SN] and Coverity.

Valgrind is a dynamic memory error detector. It works by interpreting compiled x86 programs and JIT compiling them into a reduced SPARC-like assembly, where it instruments the code, and JITs it back into x86 for execution. In the instrumentation phase, modifies the code to store an extra "validity" byte for each byte of register data or memory, and an extra addressability bit for each byte of allocated memory. The validity bits are set when they are assigned with a "valid" value (a source with the validity bits set) and cleared when assigned from a source with the bits cleared. The "addressability" bit stores whether a given address can be dereferenced; it is set on malloc and cleared on free. Valgrind will intercept the trap to Linux kernel and verify that the arguments are all valid or addressable, as appropriate for the given system call. It also tests the validity bits on branch instructions. In Valgrind, it is acceptable to copy around undefined data. An error is only thrown when the behaviour of the program is determined by an undefined value.

Coverity's checker is derived and rewritten from their earlier work, Stanford GCC[ECCH00]. This is a static checker built in to a compiler framework, using a language called "metal" to describe invalid operations, such as double calls to a locking function, in terms of the primitives in the target program.

Checker is directly built on top of a compiler framework, the Low Level Virtual Machine[LA04a]. LLVM is particularly well suited for developing new mid-level language-independent interprocedural analysis and optimizations. It also provides a type-safe bytecode language with JITter, alongside a static compiler. This compilation strategy allows for link time optimization.

The applicability of LLVM to code checking was not lost on its authors. SAFEcore[DKAL05] is a project with similar goals, but a more formal and rigorous design. Their procedure marks all potentially invalid behaviour with run-time checks, then applies static optimization to eliminate the need for the run-time checks.[DKA05]

3 The Checker

The program checker doesn't directly verify C or C++ semantics. Instead, we rely on the LLVM GCC front end to compile a program into the LLVM intermediate representation, which is then verified. Similarly, not all properties of a program are verified by the checker; LLVM includes an extensive module verifier which guarantees that all of the LLVM language assertions are held before the checker operates upon it.

Correct operation of the checker also relies on the strength of LLVMs optimizers having been applied on the input. There is no attempt made in the checker to simplify expressions or detect dead code paths. The checker will generate spurious errors if the optimizers have not been run.

There are two major categories of errors, validity errors referring to nondeterministic behaviour, and addressability errors being memory faults.

3.1 Validity

The LLVM IR includes an interesting constant value, `undef` or "undefined value" which used whenever the programs behaviour is valid regardless of the value. In practise, it is found whenever LLVM's optimizers find a case where the value is uninitialized, but not when it is indeterminate.

Nondeterministic behaviour occurs whenever an undefined value affects the control flow of the program. Since all instructions which affect control flow are enumerated as branch instructions, we enumerate all branch instructions and detect branches on `undef` argument.

- `branch on undef found.`
  An variable of undefined value will affect the control flow of the program, leading to results that are not determined by the program input.

  Example:

  ```
  extern void print(char*);

  void f()
  {
    int x;
    if (x)
      print("foo");
  ```

```
}
```

- `switch on undef found.`
  Similarly to branches, checker also finds non-determinism in switch statements.

  Example:

```
extern void print(char*);
static const int x[2][3] = {{10, 11, 12}, {13, 14, 15}};

void f()
{
  int *p = (int*)x;
  switch (p[6])
  {
    case 0:
      print("foo");
      break;
    case 1:
      print("bar");
      break;
  }
}
```

3.2 Addressability

Memory accesses concern either reads, writes, or calls of function pointers. Besides checking for `undef` pointer values, checker finds the following errors:

- `read/write/free of unallocated memory.`
  Checker has found a free instruction whose argument can not be a pointer to allocated memory. Such an instruction is always in error if executed.

  Example:

```
#include <stdlib.h>
```

```
void f() {
  char *x = (char*) 'y';
  free(x);
}
```

- `free of non-heap memory.`
  A free instruction may only free memory allocated with the malloc instruction, not with alloca nor a pointer to a variable.

  Example:

  ```
  #include <stdlib.h>

  void f() {
    char x;
    char *y = &x;
    free(y);
  }
  ```

- `write to constant memory.`
  In C, a variable declared `const` may not be modified, although the compiler will not stop you if you take a pointer and cast away constness.

  Example:

  ```
  void f() {
    char *x = "String!";
    x[0] = 's';
  }
  ```

- `write to function pointer.`
  `read from function pointer.`
  The pointer being read or written is known to be the address of a function.

  Example:

  ```
  char f() {
    char *F = f;
    return *F;
  }
  ```

- `call via non-function pointer.`

  The input code takes a pointer to a variable or allocated memory and then casts it into a function pointer and tries to call it. In some rare cases this may be intentional; a JITter will allocate memory to emit target instructions for execution. However, it is unportable and target-specific, and the behaviour is not defined by the C or C++ language standard.

  Example:

  ```
  void f() {
    char x;
    char *y = &x;
    void (*F)() = (void(*)()) y;
    F();
  }
  ```

None of these errors rely on knowledge of the type system, so the errors are found at the point of memory operation, not at a potentially harmless casting instruction. In this context, the term "function" refers to the functions visible at link time.

All of the examples compile in GCC 3.3 with no warnings, even in full warnings mode. GCC 3.4 and up note that casting between function pointer types and data pointer types is invalid.

4 Technical Novelty

Checker's most interesting work is handled by the pointer analysis mechanism to follow the propagation of pointers through the program flow. The result is that at any point in the program, checker can produce a list of all possible memory regions being aliased (a shorter list than the list of all pointers), even in a type-unsafe program.

The issue at hand is which pointers a given memory operation may be operating on. Although some operations may reduce a pointer—by masking off all of the top bits for example—could that pointer value have been calculated and used in this instruction? As written, the algorithm errs on the side of caution. Continuing our example:

```
char x;

char *f()
{
  long ptr = (long)&x;
  long x = ptr & 0x0000ffff;
  long y = ptr & 0xffff0000;
  char *p = (char*)(x + y);
  return p;
}
```

Checker never actually proves that pointer `p` is correctly pointing to our character `x`. Through data-flow, checker knows that the only pointer contributing to `p` is `&x`. We can then state is that any access through `p` either refers to `&x`, or else refers to indeterminate memory as an invalid access. At this time, checker is unable to determine whether the final pointer is in fact valid, and so makes the conservative assumption that it is.

4.1 Pointer Analysis

Checker's method of pointer analysis is flow-based, type-ignorant and whole-program approach starting with the points of creation, similar to Andersen's Analysis[And94] or Points-To Analysis using Binary Decision Diagrams[BLQ+03]. Points of creation are explicit allocations, global variables and functions, which may be dereferenced. Each of these allocations is a region from the base pointer up to a size which may not be known until execution time.

The set of instructions operating on these pointers is then calculated as a transitive closure through the instructions. Each instruction is tagged with the exhaustive set of pointers that its result may be an expression of. The algorithm to achieve this is described by three discrete passes:

1. Starting at each point of creation, iterate through the list of users and follow through its users transitively. This produces a tree where the leaf nodes are memory loads, or instructions that have no useful result such as branches, memory stores or frees. The result of any other instruction is assumed to be a function of the base pointer.

2. The load-store pairing pass finds matching store instructions for each load instruction to see if a pointer could have been stored there. If so,

the result of this load instruction is also tagged with the same pointer and we must repeat step one.

As well as stores, we also pair loads with global variables that may have been initialized with the address of another global variable.

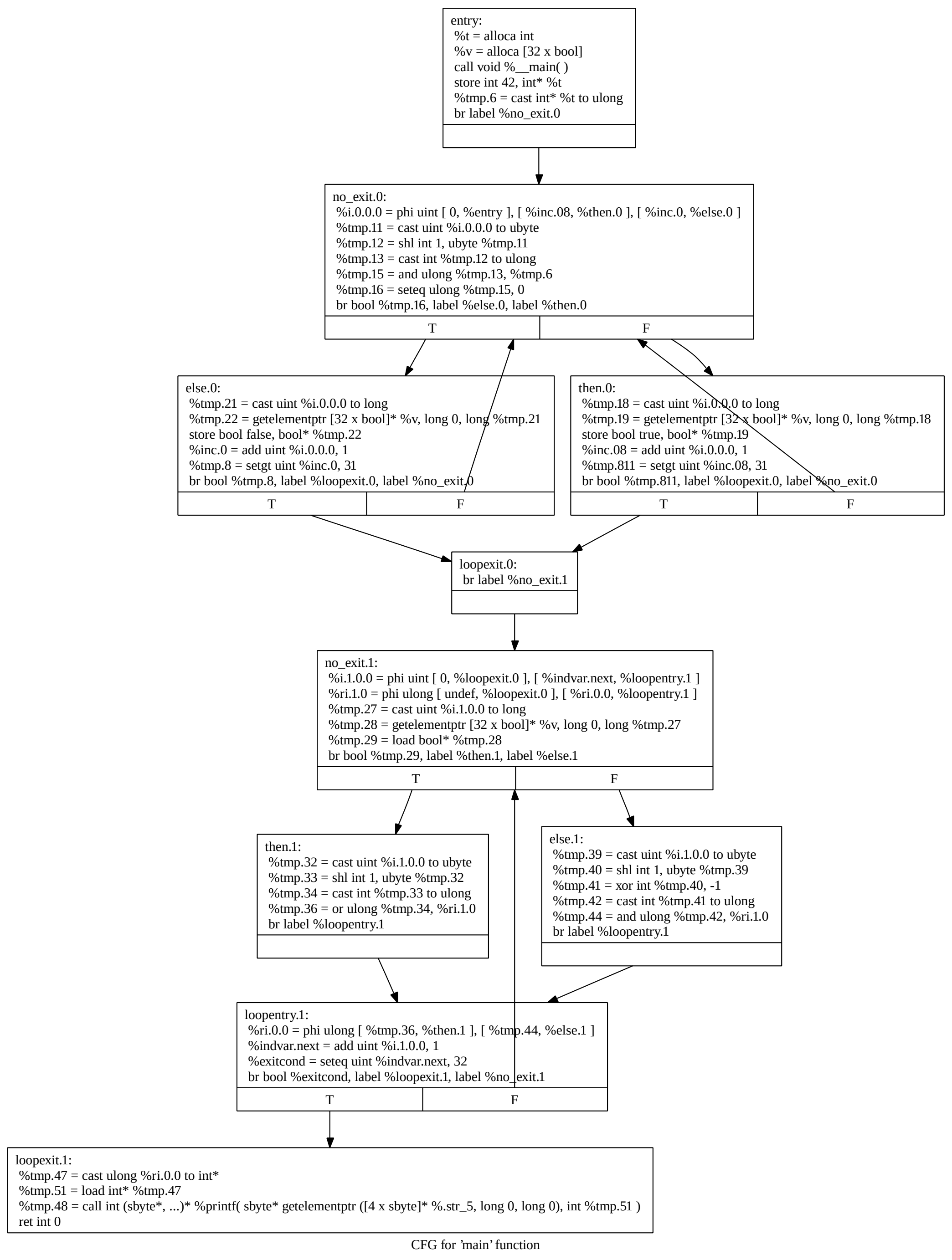
3. Finally, a pointer may be laundered through the CFG. The example program on the following page demonstrates this technique using a branch instruction to choose a basic block based on a bit in the pointer. Each basic block stores 0 or 1 to an array. There is no direct data flow from the pointer into the array. Repeat this 32 times to leak a 32-bit pointer.

   In order to detect this, we note any branch statement that is conditional on a pointer value. The pointer value is assumed to be known in the basic blocks linked to from the branch, and transitively in the blocks that it leads to, ending where the paths of the branch converge. We calculate this by taking the set of basic blocks reachable starting at the branch, and removing the intersection of basic blocks reachable by each of the branch's conditions. The result is the set of blocks that could expose the pointer value.

   Within these blocks, any store to memory could potentially be leaking the pointer, even the store of a constant. As such all stores are tagged. More improbable is that the result of all loads must also be tagged. One example would be a 32 levels of nested if statements uniquely identifying each possible value of a 32-bit pointer, then dereferencing that constant value.

   After that, newly defined SSA variables inside these basic blocks are tagged. The successor blocks are checked to see if there are any ϕ-nodes leading back into the set. If so, they are tagged as well. Now that these new instructions have been added to the closure, we must repeat the algorithm starting at step one.

Through this algorithm, the checker solves for the set of all possible pointer values involved in the execution of any given instruction.



```
entry:
 %t = alloca int
 %v = alloca [32 x bool]
 call void %__main( )
 store int 42, int* %t
 %tmp.6 = cast int* %t to ulong
 br label %no_exit.0
```

```
no_exit.0:
 %i.0.0.0 = phi uint [ 0, %entry ], [ %inc.08, %then.0 ], [ %inc.0, %else.0 ]
 %tmp.11 = cast uint %i.0.0.0 to ubyte
 %tmp.12 = shl int 1, ubyte %tmp.11
 %tmp.13 = cast int %tmp.12 to ulong
 %tmp.15 = and ulong %tmp.13, %tmp.6
 %tmp.16 = seteq ulong %tmp.15, 0
 br bool %tmp.16, label %else.0, label %then.0
```

```
else.0:
 %tmp.21 = cast uint %i.0.0.0 to long
 %tmp.22 = getelementptr [32 x bool]* %v, long 0, long %tmp.21
 store bool false, bool* %tmp.22
 %inc.0 = add uint %i.0.0.0, 1
 %tmp.8 = setgt uint %inc.0, 31
 br bool %tmp.8, label %loopexit.0, label %no_exit.0
```

```
then.0:
 %tmp.18 = cast uint %i.0.0.0 to long
 %tmp.19 = getelementptr [32 x bool]* %v, long 0, long %tmp.18
 store bool true, bool* %tmp.19
 %inc.08 = add uint %i.0.0.0, 1
 %tmp.811 = setgt uint %inc.08, 31
 br bool %tmp.811, label %loopexit.0, label %no_exit.0
```

```
loopexit.0:
 br label %no_exit.1
```

```
no_exit.1:
 %i.1.0.0 = phi uint [ 0, %loopexit.0 ], [ %indvar.next, %loopentry.1 ]
 %ri.1.0 = phi ulong [ undef, %loopexit.0 ], [ %ri.0.0, %loopentry.1 ]
 %tmp.27 = cast uint %i.1.0.0 to long
 %tmp.28 = getelementptr [32 x bool]* %v, long 0, long %tmp.27
 %tmp.29 = load bool* %tmp.28
 br bool %tmp.29, label %then.1, label %else.1
```

```
then.1:
 %tmp.32 = cast uint %i.1.0.0 to ubyte
 %tmp.33 = shl int 1, ubyte %tmp.32
 %tmp.34 = cast int %tmp.33 to ulong
 %tmp.36 = or ulong %tmp.34, %ri.1.0
 br label %loopentry.1
```

```
else.1:
 %tmp.39 = cast uint %i.1.0.0 to ubyte
 %tmp.40 = shl int 1, ubyte %tmp.39
 %tmp.41 = xor int %tmp.40, -1
 %tmp.42 = cast int %tmp.41 to ulong
 %tmp.44 = and ulong %tmp.42, %ri.1.0
 br label %loopentry.1
```

```
loopentry.1:
 %ri.0.0 = phi ulong [ %tmp.36, %then.1 ], [ %tmp.44, %else.1 ]
 %indvar.next = add uint %i.1.0.0, 1
 %exitcond = seteq uint %indvar.next, 32
 br bool %exitcond, label %loopexit.1, label %no_exit.1
```

```
loopexit.1:
 %tmp.47 = cast ulong %ri.0.0 to int*
 %tmp.51 = load int* %tmp.47
 %tmp.48 = call int (sbyte*, ...)* %printf( sbyte* getelementptr ([4 x sbyte]* %.str_5, long 0, long 0), int %tmp.51 )
 ret int 0
```

Figure 1: Break up a pointer into an array of booleans and reassemble it

5 Future Work

The implementation is missing some critical features preventing it from being used to verify real world software. Primarily, it lacks interprocedural analysis. Note that this is not a theoretical limitation—an inliner can rewrite a program without function calls—nor is it a limitation due to the use of LLVM—LLVM is famous for performing extensive interprocedural optimizations—but merely a limit of the current implementation.

Checker's analysis passes also suffer from other, smaller, deficiencies:

1. The validity pass is unable to detect an out of bounds access to an array, even if the bounds and index are both known at compile time. Currently, any value taken from a memory load is assumed to be valid, even if there is no matching store at all!

   The proper way to fix this is to expose the pointer analysis to LLVMs optimizers through the provided AliasAnalysis interface, so that LLVM can find such cases and substitute them for `undef`.

2. Due to its optimization-centric design, LLVM naturally removes partially undefined values. For example:

   ```
   int f() {
     int x;
     x |= 1;
     return x;
   }
   ```

   is optimized into:

   ```
   int %f() {
   entry:
           ret int -1
   }
   ```

   While this substitution is always valid for an optimizer, it replaces undefined behaviour with defined behaviour, circumventing checker's ability to detect the non-determinism in the original code. However, LLVM will generate code based on the optimized version, and so, the resulting program will not be non-deterministic. In this sense, checker's failure to warn about this error could be considered correct, as it is operating on the compiler's interpretation of the program, not the language standard's interpretation. Nevertheless, LLVM could be modified to change its treatment of `undef`.

3. The pointer analysis is region-based, and unable to differentiate between accesses of different fields in an array or structure.

4. The pointer analysis could be made less conservative in step three by eliminating store instructions that are identical in all branches and guaranteed to execute. This is made slightly more difficult because the order of multiple store instructions storing to the same region needs to be compared.

5. Region-based analysis could become a more fine-grained field-based analysis.

Even the best analysis process is limited by its interface with the user. One way to improve checker is to make it interactive, allowing the user to see what the aliasing set of a variable is at a particular point. A more advanced version of the same would employ LLVMs JITter to allow a mixed static/dynamic analysis where the user could enter values for the registers and execute for a few instructions to test a hypothesis.

Finally, checker needs to be able to prune pointers from the aliasing set when a conditional is encountered.

6 Conclusion

Finding simple common errors proves possible with relatively little work. We have found that checker can detect coding errors and offers opportunities for further extension, but is not yet capable of properly analyzing real-world software.

References

[ALB$^+$03] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set

Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.
http://llvm.org/pubs/2003-10-01-LLVA.html

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.
http://repository.readscheme.org/ftp/papers/topps/D-203.pdf

[BLQ+03] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*, pages 103–114. ACM Press, 2003.

[DKA05] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Enforcing Alias Analysis for Weakly Typed Languages. Technical Report #UIUCDCS-R-2005-2657, Computer Science Dept., Univ. of Illinois, Nov 2005.
http://llvm.org/pubs/2005-11-SAFECodeTR.html

[DKAL05] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *Trans. on Embedded Computing Sys.*, 4(1):73–111, 2005.
http://llvm.org/pubs/2005-02-TECS-SAFECode.html

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. *4th Symposium on Operating Systems Design & Implementation*, 2000.
http://www.stanford.edu/ engler/mc-osdi.ps

[LA04a] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
http://llvm.org/pubs/2004-01-30-CGO-LLVM.html

[LA04b] Chris Lattner and Vikram Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.
http://llvm.org/pubs/2004-09-22-LCPCLLVMTutorial.html

[Lat06] Chris Lattner. LLVM Assembly Language Reference Manual, May 2006.
http://llvm.org/docs/LangRef.html

[SN] Julian Seward and Nick Nethercote. Valgrind, an open-source memory debugger for x86-GNU/Linux.
http://www.valgrind.org/