Jianzhou Zhao
Steve Zdancewic
Milo Martin
University of Pennsylvania

Santosh Nagarakatte
University of Pennsylvania/Rutgers University
Genesis

Using LLVM since 2007

Pointer-based checking for memory Safety in the LLVM IR

Can we trust the implementations of the instrumentation?

[Nagarakatte, et al. PLDI ’09]
Can we trust the implementation of the compiler?
Compilers are not Always Correct

[Yang, et al. PLDI ’11]
Compilers are not Always Correct

[Yang, et al. PLDI '11]

Csmith

Random test-case generation

Source Programs

GCC

79 bugs: 25 critical

LLVM

202 bugs

{8 other C compilers}

325 bugs in total
Compilers are not Always Correct

[Yang, et al. PLDI '11]

Csmith

Random test-case generation

Source Programs

Compiler

Transformed Programs

Bug kinds:
- Compile-time crash
- Output wrong-code
How can ensure that the compiler implementations are free of errors?
Eliminating Compiler Bugs

- Extensive testing, comprehensive test suites
- Write the compiler in a proof assistant

Compiler
The CompCert Project

- A realistic C compiler
- Proved semantics-preservation in Coq

http://compcert.inria.fr

[Extraction from the formalization]

[Leroy, et al.]

CompCert C → Verified Transformations → PowerPC, ARM, and x86

PowerPC, ARM, and x86

CompCert C

CompCert

Caml

Santosh Nagararakatte – Vellvm - 2012
Verified Compilers are Robust

“The apparent **unbreakability of CompCert** supports a strong argument that **developing compiler optimizations within a proof framework**, where safety checks are explicit and machine-checked, **has tangible benefits** for compiler users.”
Can we build LLVM within a Proof Assistant?
Type System and SSA

Syntax

Memory Model

Operational Semantics

OCaml-LLVM Bindings

Printer

Parser

Extract OCaml AST from Coq

OCaml AST

LLVM IR

Pass

LLVM IR

Pass

LLVM IR

...
Vellvm’s Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
Vellvm’s Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
- Reverse-engineering

Santosh Nagarakatte – Vellvm - 2012
Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
    - Typed, SSA
    - Non-deterministic, with high-level memory model
    - Formalized with proofs in mind

Santosh Nagarakatte – Vellvm - 2012
Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
  - Typed, SSA
  - Non-deterministic, with high-level memory model
  - Formalized with proofs in mind

- Tools for interacting with LLVM
Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
  - Typed, SSA
  - Non-deterministic, with high-level memory model
  - Formalized with proofs in mind
- Tools for interacting with LLVM
- Application of the semantics
  - Verified SoftBound Transformation
  - Verified Mem2reg Transformation
Outline

• Background

• Semantics of the LLVM IR

• Applications of the semantics
  • Tool chain
  • Verified SoftBound transformation
  • Verified Mem2reg optimization

• Lesson learned & Conclusion
The Coq Proof Assistant

- A programming language for proofs
- The expressive core language: CIC
- Extract programs from Coq

Santosh Nagarakatte – Vellvm - 2012
• Formal semantics
• Verified transformations
Vellvm
%ST = type {i10,[10 x i8*]}

define %ST @foo(%ST %st₀, i8* %ptr)
{
    entry:
    %p = malloc %ST, i32 1
    store %ST %st₀, %p
    %r = gep %ST* %p, i32 0, i32 0
    store i10 648, %r
    %s = gep %ST* %p, i32 0, i32 1, i32 0
    store i8* %ptr, %s
    br undef %loop %succ

    loop:
    %x = phi i32 [%z, %loop], [0, %entry]
    %z = phi i32 [%x, %loop], [1, %entry]
    %b = icmp leq %x %z
    br %b %loop %succ

    succ:
    %st₁ = load (%ST*) %p
    free (%ST*) %p
    ret %ST %st₁
}

- LLVM assembly language
- Typed
- SSA
- First-class high-level data type
  - Store/Load
  - GEP (accessing sub-fields)

Vellvm formalizes domination analysis to reason about SSA

LLVM operational semantics is non-deterministic.

Santosh Nagarakatte – Vellvm - 2012
Non-determinism with Undef values
An LLVM value is a set of values.

- All 8-bit integers
- i8 undef \(\equiv\) \{0 – 255\}
- i8 1 \(\equiv\) \{1\}
- A singleton set

\(\%x = \text{or i8 undef 1}\)

or

i8 \(\%x \equiv\) \{1, 3, 5, …, 255\}
An LLVM value is a set of values.

All 8-bit integers

i8 undef ≡ \{ 0 – 255\}

i8 1 ≡ \{1\}

A singleton set

or

i8 %x = or i8 undef 1

i8 %x ≡ \{1, 3, 5, \ldots, 255\}
An LLVM value is a set of values.

- All 8-bit integers
- i8 undefined $\equiv \{0 - 255\}$
- i8 1 $\equiv \{1\}$
- A singleton set

\[
\%x = \text{or} \ i8 \text{ undef} \ 1
\]

or

\[
i8 \ \%x \equiv \{1, 3, 5, \ldots, 255\}
\]

Santosh Nagarakatte – Vellvm - 2012
An LLVM value is a set of values.

All 8-bit integers

i8 undef \equiv \{ 0 \text{ – } 255 \}

i8 1 \equiv \{1\}

A singleton set

\%x = or i8 undef 1

or

i8 \%x \equiv \{1, 3, 5, \ldots, 255\}

All 8-bit odd integers

Santosh Nagarakatte – Vellvm - 2012
One more example...

All 8-bit integers

i8 undef \equiv \{0 - 255\}

i8 1 \equiv \{1\}

A singleton set

%x = or i8 undef 1

or

i8 %x \equiv \{1, 3, 5, ..., 255\}

xor

%y = xor i8 %x %x

i8 %y \equiv \{0\} ??

Santosh Nagarakatte – Vellvm - 2012
One more example...

- `%x = or i8 undef 1`
- `i8 undef ≡ \{0 – 255\}`
- `i8 1 ≡ \{1\}`
- A singleton set

or

- `%y = xor i8 %x %x`
- `i8 %x ≡ \{1, 3, 5, \ldots, 255\}`

Enabling more optimizations than returning \{0\}

- `i8 %y ≡ \{0, 2, 4, \ldots, 254\}`

Santosh Nagarakatte – Vellvm - 2012
Non-deterministic Branching

\[ l_0: \\
... \\
br \text{ undefined} \ l_1 \ l_2 \]

??
Non-deterministic Branching

\[ l_0: \]
\[ ... \]
\[ br \ undefined \ l_1 \ l_2 \]

\[ l_1: \]
\[ ... \]
\[ ... \]

\[ l_2: \]
\[ ... \]
\[ ... \]
Memory Model

- Reason about memory operations
  - Without being specific to the platform/memory manager

Being pragmatic, we reuse and extend CompCert’s memory model
The LLVM Memory Model

Memory operations with high-level values

Santosh Nagarakatte – Vellvm - 2012
The LLVM Memory Model

Byte-oriented representation

Memory operations with high-level values

Physical Subtyping

Santosh Nagarakatte – Vellvm - 2012
The LLVM Memory Model in Vellvm

Accessing memory with incompatible types returns undefs

Byte-oriented representation

- i10
- i8*
- i8*
- i8*
- i8*
- i8*
- i8*
- i8*
- i8*

Dynamically typed

Flattened values and memory accesses

High-level value

- field₁
- field₂

Flattening
- store
- load

Construction

Santosh Nagarakatte – Vellvm - 2012
Modeling Undefined Behaviors
Sources of Undefined Behaviors

1. Loading from uninitialized memory locations
   - `%p = alloca i4`
   - `%r = load (i4*) %p`

2. Uninitialized variables
   - `int x;`
   - `y = 1 + x;`

3. Loading with mismatched types

4. Free invalid pointers

5. Access dangling pointers

6. Access out-of-bound addresses

7. Invalid indirect calls

Santosh Nagarakatte – Vellvm - 2012
Sources of Undefined Behaviors

- Compilers are conservative to not introduce undefined behaviors
- LLVM allows more aggressive optimizations
Undefs with LLVM

**Type 1**

- Loading from uninitialized memory locations
  - \(\%p = \text{alloca } \text{i4}\)
  - \(\%r = \text{load } (\text{i4}* ) \%p\)

- Uninitialized variables
  - \(\%y = \text{add } \text{i32} 1 \text{ undef}\)

**Type 2**

- Free invalid pointers
- Access dangling pointers
- Access out-of-bound addresses
  - cause segmentation fault, bus error, ...

Santosh Nagarakatte – Vellvm - 2012
Undeﬁned Values

Loading from uninitialized memory locations

%\texttt{p} = \texttt{alloca} \texttt{i}4

%\texttt{r} = \texttt{load} (\texttt{i}4\*) %\texttt{p}

Undefined values

Uninitialized variables

%\texttt{y} = \texttt{add} \texttt{i}32 \texttt{1} \texttt{undef}

Loading with mismatched types

Undefined Behaviors

Free invalid pointers

- Undefined values
  - a set of values
  - aggressive optimizations

Invalid indirect calls

Santosh Nagarakatte – Vellvm - 2012
Nondeterministic Operational Semantics

An LLVM runtime value is a set of values

LLVM_{ND}: \textit{config} \vdash S \rightarrow S'

small-step

Relational: non-deterministic relations between program states

Otherwise, the semantics is straightforward and tractable.
### Partiality

<table>
<thead>
<tr>
<th>Free invalid pointers</th>
<th>Access dangling pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>free (i8*) NULL</code></td>
<td><code>free (i8*) %p</code></td>
</tr>
<tr>
<td><code>free (i8*) %p</code></td>
<td><code>%r = load (i8*) %p</code></td>
</tr>
<tr>
<td><code>free (i8*) %p</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

Stuck \((\text{config}, S) = \text{BadFree} (\text{config}, S))\)  
\(\vee \text{BadLoad} (\text{config}, S))\)  
\(\vee \text{BadStore} (\text{config}, S))\)  
\(\vee ...\)  
\(\vee ...\)

Access out-of-bound addresses

---

Santosh Nagarakatte – Vellvm - 2012

---

%p = malloc i8
%q = gep %p, i32 255
store i8 0, %q
Preservation and Progress

S is well-typed and in the SSA form:

\[
\text{If } \text{config} \vdash S
\]

Preservation:

\[
\text{If } \text{config} \vdash S, \text{ and } \text{config} \vdash S \rightarrow S', \text{ then } \text{config} \vdash S'.
\]

Progress:

\[
\text{If } \text{config} \vdash S, \text{ then } S \text{ terminates or } \text{Stuck(config, S)} \text{ or exists } S', \text{config} \vdash S \rightarrow S'.
\]
Tools & Applications
Extracted Interpreter

Syntax

Type System and SSA

Memory Model

Operational Semantics

Extract Interpreter from Coq

Comp

Extracted Interpreter

Referenced Interpreter

Test Suite

LLVM IR

Pass

LLVM IR

Santosh Nagarakatte – Vellvm - 2012
Verified SoftBound
SoftBound

C Source Code

LLVM

An instrumentation pass in LLVM
Detect spatial memory safety violation (buffer overflow, …)

 LLVM IR

Nagarakatte, et al. PLDI ’09

http://www.cis.upenn.edu/acg/softbound/

Santosh Nagarakatte – Vellvm - 2012
%p = malloc i8

%q = gep %p, i32 255

store i8 0, %q

%p = malloc i8

%p_{\text{base}} = gep %p, i32 0

%p_{\text{bound}} = gep %p, i32 1

%q = gep %p, i32 255

%q_{\text{base}} = %p_{\text{base}}

assert %q_{\text{base}} \leq %q \land %q+1 < %q_{\text{bound}}

store i8 0, %q

Santosh Nagarakatte – Vellvm - 2012
```
%p = malloc i8

Maintain base and bound for all pointers

%q = gep %p, i32 255

Propagate metadata on assignment

store i8 0, %q

Check if a pointer is within its base and bound when being accessed

%p = malloc i8

%p_base = gep %p, i32 0

%p_bound = gep %p, i32 1

%q_base = %p_base

%q_bound = %p_bound

assert %q_base <= %q \% %q+1 < %q_bound

store i8 0, %q

Raise exception for the spatial memory violation

Pointers in temporaries

C Source Code ➔ LLVM IR ➔ SoftBound ➔ LLVM IR ➔ LLVM Optimizations ➔ Target
```

Santosh Nagarakatte – Vellvm - 2012
User memory

Disjoint metadata

Cannot be changed by any spatial memory violation!!!
Extracting an executable SoftBound pass from the formal development in Coq.
Partiality

Free invalid pointers

free (i8*) NULL
free (i8*) %p
free (i8*) %p

Access out-of-bound addresses

store i8 0, %q

Access dangling pointers

free (i8*) %p
%r = load (i8*) %p

Stuck (config, S) = BadFree (config, S) ∨ BadLoad (config, S) ∨ BadStore (config, S)

SpatialMemoryViolation(config, S)

Santosh Nagarakatte – Vellvm - 2012
Correctness of SoftBound

If $\text{config}_1 \models S_1$, 
$\text{SoftBound}(\text{config}_1, S_1) = (\text{config}_2, S_2)$, and 
$\text{config}_2 \models S_2 \rightarrow^* S'_2$, 
then $\sim \text{SpatialMemoryViolation}(\text{config}_2, S'_2)$

A transformed program
- has no spatial memory safety violation
Correctness of SoftBound

If $\text{config}_1 \models S_1$, 
$\text{SoftBound}(\text{config}_1, S_1) = (\text{config}_2, S_2)$, and 
$\text{config}_2 \models S_2 \rightarrow S'_2$, 
then $\sim \text{SpatialMemoryViolation}(\text{config}_2, S'_2)$ \land 
exists $S'_1$, 
$\text{config}_1 \models S_1 \rightarrow S'_1 \land (\text{config}_1, S'_1) \approx (\text{config}_2, S'_2)$.

A transformed program
• has no spatial memory safety violation
• preserves the semantics of its original program

Santosh Nagarakatte – Vellvm - 2012
Bugs Found in the Original SoftBound

- Incorrect metadata initialization

<table>
<thead>
<tr>
<th>pointer</th>
<th>base</th>
<th>bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>undef</td>
<td>undef</td>
<td>undef</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL+ sizeof(typ)</td>
</tr>
</tbody>
</table>

Unsafe to access!

Fix

<table>
<thead>
<tr>
<th>pointer</th>
<th>base</th>
<th>bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>undef</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Invalid access to these pointers

Santosh Nagarakatte – Vellvm - 2012
Competitive Runtime Overhead

The performance of extracted SoftBound is competitive with the non-verified original.
Written with verification in mind

- Simplify both design and proofs
- Increase robustness

A verified and more robust SoftBound with competitive performance in practice.

Santosh Nagarakatte – Vellvm - 2012
Optimization with maximum performance benefit
Performance-critical Optimization Pass in LLVM

Mem2reg is the optimization with that provides the maximum benefit
Mem2reg in LLVM

The LLVM IR in the trivial SSA form

\begin{itemize}
  \item \texttt{x=0 if } y \leq 0
  \item \texttt{x=1 if } y > 0
\end{itemize}

\begin{verbatim}
int x = 0;
if(y>0) x=1;
return x;
\end{verbatim}

\begin{verbatim}
\texttt{\%y \leq 0}
\texttt{\%y > 0}
\texttt{\%b = \%y > 0}
\texttt{br \%b \%l_2 \%l_3}
\texttt{\%y > 0}
\texttt{store 1 \%p}
\texttt{br \%l_3}
\texttt{\%x = load \%p}
\texttt{ret \%x}
\end{verbatim}
Mem2reg in LLVM

C Code

```c
int x = 0;
if(y>0) x=1;
return x;
```

• x=0 if y ≤ 0
• x=1 if y > 0

The LLVM IR in the trivial SSA form

```llvm
l1:
    %p = alloca i32
    store 0 %p
    %b = %y > 0
    br %b %l2 %l3

l2:
    store 1 %p
    %y > 0
    br %l3

l3:
    %x = load %p
    ret %x
```

The LLVM IR in the minimal SSA form

```llvm
l1:
    %b = %y > 0
    br %b %l2 %l3

l2:
    br %l3

l3:
    %x = %c 1,%l2 %0,%l1
    %y > 0, %y ≤ 0
    ret %x
```
Verified mem2reg

- Wegman et al’s SSA construction is commonly used
- Intermediate stage breaks the SSA invariant
- Cannot be proved easily

We designed a new mem2reg that preserves SSA at each step
The runtime speedup of vmem2reg

Vmem2reg does not generate the pruned SSA form

- Extracted vmem2reg
- LLVM's mem2reg
Lessons & Conclusions

- Formal semantics of LLVM IR
  - Meta-theoretic results & tools
  - Verified SoftBound and verified Mem2reg
  - Foundations for a large number of tools/research

- Developer involvement is useful
  - Interesting to explore tradeoffs
    - Easier to prove vs performance vs compilation time
Looking Ahead

Foundation for verifying the industry-strength modern compiler --- LLVM

- Verified LLVM frontends
- Verified LLVM backends
- Reasoning about LLVM programs
- Verified optimizations
- Verified program analysis
- ...
Want to try?

http://www.cis.upenn.edu/~jianzhou/Vellvm

email us for the latest LLVM version