Reconciling High-Level Optimizations and Low-Level Code in LLVM

Seoul National Univ.

University of Utah

MPI-SWS

Microsoft Research

Juneyoung Lee
Chung-Kil Hur

Ralf Jung

Zhengyang Liu
John Regehr

Nuno P. Lopes
Overview
Overview

Allows access via int-to-ptr cast

C

Low-level Code

PC
Overview

C
Low-level Code

LLVM IR

High-level Optimizations

P_C

P_I

P'_I

Allows access via int-to-ptr cast
Overview

Allows access via int-to-ptr cast

Assumes no one can access my local vars

C
Low-level Code

PC

LLVM IR

High-level Optimizations

P_{IR}

P'_{IR}
Overview

Allows access via int-to-ptr cast

Assumes no one can access my local vars
Finding a Good Memory Model

- A memory model specifies the behavior of memory operations
- As a result, it determines
  1. Which low-level programs are valid
  2. Which high-level assumptions are valid
- A good memory model should make valid both
  1. Common low-level programs
  2. Common high-level assumptions
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
Memory ≠ Byte Array

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
Memory ≠ Byte Array

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

constant prop.
Memory ≠ Byte Array

We use C syntax for LLVM IR code for readability

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

constant prop.

Memory ≠ Byte Array

Memory:

0x0

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
Memory ≠ Byte Array

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Memory:

```
0x0  0x100
```

constant prop.

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}

Memory ≠ Byte Array

constant prop.
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);  
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
Memory ≠ Byte Array

Problem
q can be accessed from p by pointer arithmetic

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
Abstract Memory Explains Optimizations

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```

**Provenance**

Memory:

```
<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x100</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x101</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Constant Propagation**
Abstract Memory Explains Optimizations

Provenance

Memory:

\[
\begin{array}{c}
\text{p: -} \\
0x0 \\
0x100 \\
0x101 \\
\text{q: 0}
\end{array}
\]

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

constant prop.

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
Abstract Memory Explains Optimizations

Provenance

Memory:

\[(p, 0x100), (q, 0x101)\]

\[
\begin{align*}
\text{char } & p[1], q[1] = \{\emptyset\}; \\
\text{int } & ip = (\text{int})(p+1); \\
\text{int } & iq = (\text{int})q; \\
\text{if (iq == ip)} & \\
\text{*(p+1)} & = 10; \\
\text{print}(q[0]); \\
\end{align*}
\]

\[
\begin{align*}
\text{char } & p[1], q[1] = \{\emptyset\}; \\
\text{int } & ip = (\text{int})(p+1); \\
\text{int } & iq = (\text{int})q; \\
\text{if (iq == ip)} & \\
\text{*(p+1)} & = 10; \\
\text{print}(\emptyset); \\
\end{align*}
\]
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
Abstract Memory Explains Optimizations

Principles of UB

1. Compilers assume input programs never raise UB
2. Programmers should not write programs raising UB

Undefined Behavior because $p \neq q$
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);
}
```
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

Constant propagation

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

Integer equality

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

Cast elimination
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

character equality property.
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p + 1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p + 1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p + 1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}

int eq. prop.

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
Miscompilation with Int-Ptr Casting

char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*) (int) (p+1) = 10;
    print(q[0]);
}

cast elim.

int. eq. prop.

char p[1], q[1] = {0};
int ip = (int) (p+1);
int iq = (int) q;
if (iq == ip) {
    *(char*) iq = 10;
    print(q[0]);
}

cast elim.

constant prop.

char p[1], q[1] = {0};
int ip = (int) (p+1);
int iq = (int) q;
if (iq == ip) {
    *(char*) (p+1) = 10;
    print(q[0]);
}
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)ip = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1)=10;
    print(q[0]);
}
```
Miscompilation with Int-Ptr Casting

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
We found this miscompilation bug in both LLVM & GCC
Miscompilation with Int-Ptr Casting

We found this miscompilation bug in both LLVM & GCC

Goal of this paper
Finding a good memory model for pointer ↔ integer casting
Problems &
Our Solutions
Problem 1
Pointer → Integer Casting?

\[(\text{int})p\]
Pointer → Integer Casting?

\[(p, 0x100)\]

\[(\text{int})p\]
Pointer → Integer Casting?

(p, 0x100) → (int)p
1. Carry Provenance
Pointer $\rightarrow$ Integer Casting?

1. Carry Provenance
2. Drop Provenance
Pointer $\rightarrow$ Integer Casting?

1. Carry Provenance

2. Drop Provenance
Pointer $\rightarrow$ Integer Casting?

1. Carry Provenance $\times$
2. Drop Provenance $\checkmark$
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i \ : \ j) \quad \rightarrow \quad k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i==j \ ? \ i : j) \]

true

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i : j) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i : j) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

k = (i==j ? i : j)

false

k = j
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i : j) \]

\[ k = j \]
1. Carry Provenance

\[ k = (i == j ? i : j) \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j ? i : j) \]

\[ (p, 0x100) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i : j) \]

\[ (p, 0x100) \]

\[ (q, 0x100) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j ? i : j) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j ? i : j) \]

\[ k = j \]
Carry Provenance: 
Integer Optimization Problem

1. Carry Provenance

\[ k = (i == j \ ? \ i : j) \]

\[ k = j \]

\[ (p, 0x100) \]

\[ true \]

\[ (q, 0x100) \]

\[ (q, 0x100) \]
Carry Provenance: Integer Optimization Problem

Problem

Integer optimizations may change provenance

\[ k = (i == j \ ? \ i : j) \]

\[ k = j \]
Carry Provenance: Integer Optimization Problem

2. Drop Provenance

\[ k = (i == j ? i : j) \]

\[ k = j \]
Carry Provenance:
Integer Optimization Problem

2. Drop Provenance

\[ k = (i==j \ ? \ i \ : \ j) \]

\[ k = j \]
Problem 2
Integer → Pointer Casting?

Memory: \[ \text{char p[1]} \]
\[ 0x100 \]

\[ (\text{int}*)0x100 \]
Integer → Pointer Casting?

1. Always Empty
Integer → Pointer Casting?

Memory: char p[1]

0x100

Ø,0x100

(int*)0x100

1. Always Empty

2. Depending on the Memory Layout
Integer → Pointer Casting?

Memory: 0x100

1. Always Empty
2. Depending on the Memory Layout
Integer → Pointer Casting?

Memory: char p[1] 0x100

1. Always Empty
2. Depending on the Memory Layout
Integer → Pointer Casting?

Memory: \((\text{char } p[1])\)

0x100

- (\(\emptyset, 0x100\))
- (\(p, 0x100\))
- (\(*, 0x100\))

1. Always Empty
2. Depending on the Memory Layout
3. Always Full
Integer → Pointer Casting?

Memory: $\text{char p[1]} \rightarrow 0x100$

1. Always Empty
2. Depending on the Memory Layout
3. Always Full

$(\phi, 0x100)$ $(p, 0x100)$ $(*, 0x100)$
Integer → Pointer Casting?

Memory: char p[1] → 0x100

1. Always Empty
   (Ø, 0x100) → X

2. Depending on the Memory Layout
   (p, 0x100) → X

3. Always Full
   (*, 0x100) → ✔
Empty Provenance: 
Pointer – Integer Round Trip

1. Always Empty

Memory:

\[
\begin{align*}
\text{char } p[1] \\
0x100 \\
i &= (\text{int})p \\
p2 &= (\text{char}*)i \\
*p2 &= 10
\end{align*}
\]
Empty Provenance:
Pointer – Integer Round Trip

1. Always Empty

Memory:

\[
\text{char } p[1] \\
0x100
\]

\[
i = (\text{int})p \\
p2 = (\text{char}*)i \\
*p2 = 10
\]
Empty Provenance: Pointer – Integer Round Trip

1. Always Empty

Memory:

\[
\text{char } p[1]
\]

\[
0x100
\]

\[
i = (\text{int})p
\]

\[
p2 = (\text{char}*)i
\]

\[
*p2 = 10
\]
Empty Provenance: Pointer – Integer Round Trip

1. Always Empty

Memory:

```
char p[1]
```

```
0x100
```

```
0x100
(∅,0x100)
```

```
i = (int)p
p2 = (char*)i
*p2 = 10
(p,0x100)
```
Empty Provenance: Pointer – Integer Round Trip

1. Always Empty

Memory:

\[ \text{char } p[1] \]

\[ 0x100 \]

\[ (\emptyset, 0x100) \]

\[ 0x100 \]

\[ i = (\text{int})p \]

\[ p2 = (\text{char}*)i \]

\[ *p2 = 10 \]

UB
Empty Provenance: Pointer – Integer Round Trip

Problem
Common program patterns raise UB

Memory: char p[1]
0x100

0x100
(∅,0x100)

i = (int)p

p2 = (char*)i

*p2 = 10

UB
Empty Provenance: Pointer – Integer Round Trip

3. Always Full

Memory:

```
char p[1]
```

```
i = (int)p
p2 = (char*)i
*p2 = 10
```

UB

0x100

(∅,0x100)
Empty Provenance: Pointer – Integer Round Trip

3. Always Full

Memory: char p[1]

0x100

0x100

(*,0x100)

\[
i = (\text{int})p
\]

\[
p2 = (\text{char}*)i
\]

\[
*p2 = 10
\]

UB

UB
Empty Provenance: Pointer – Integer Round Trip

3. Always Full

Memory:

\[ i = (\text{int})p \]
\[ p2 = (\text{char}*)i \]
\[ *p2 = 10 \]
Integer → Pointer Casting?

Memory: char p[1] → 0x100

1. Always Empty
2. Depending on the Memory Layout
3. Always Full

(int*)0x100

(Ø,0x100)  (p,0x100)  (*,0x100)
Integer $\rightarrow$ Pointer Casting?

Memory: $\text{char p[1]}$ ↔ $0x100$

$(\text{int*})0x100$

1. Always Empty
2. Depending on the Memory Layout
3. Always Full
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

Memory:

char *p = malloc(1)
q = (int*)0x100

char p[1]
0x100

q = (int*)0x100
char *p = malloc(1)
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout:

```c
char *p = malloc(1);
q = (int*)0x100;
```

Memory:

```
char p[1]
0x100
```

```
q = (int*)0x100
char *p = malloc(1)
```
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout:

```
char *p = malloc(1)
q = (int*)0x100
```

Memory:

```
0x100
```

```
char p[1]
```

```
q = (int*)0x100
char *p = malloc(1)
```

```
(p, 0x100)
```
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

Memory: ________ → 0x100

```
char *p = malloc(1);
q = (int*)0x100
(p, 0x100)
```

```
q = (int*)0x100
char *p = malloc(1)
```
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

Memory:

\[
\begin{align*}
\text{char *} p & = \text{malloc}(1) \\
q & = (\text{int}*)0x100
\end{align*}
\]

\[
\begin{align*}
(p, 0x100) & \\
(\emptyset, 0x100) & \\
q & = (\text{int}*)0x100 \\
\text{char *} p & = \text{malloc}(1)
\end{align*}
\]
Depending on the Memory Layout: Reordering

2. Depending on the Memory Layout

**Problem**
Movement of casts, or functions including them, is restricted

char *p = malloc(1)
q = (int*)0x100
(p,0x100)

q = (int*)0x100
char *p = malloc(1)
Depending on the Memory Layout: Reordering

3. Always Full

Memory:

\[
\begin{align*}
\text{char } p[1] & \\
0x100 & \rightarrow (\emptyset, 0x100)
\end{align*}
\]

\[
\begin{align*}
\text{char } *p & = \text{malloc}(1) \\
q & = (\text{int}*)0x100 \\
\text{char } *p & = \text{malloc}(1)
\end{align*}
\]
Depending on the Memory Layout: Reordering

3. Always Full

Memory:

```
char p[1]
```

0x100

\((*,0x100)\)

\[
\text{char } *p = \text{malloc}(1) \\
q = (\text{int}*)0x100
\]

\[
\text{char } *p = \text{malloc}(1)
\]
Problem 3
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting

```c
char p[1] = {0};
f();
print(p[0]);
```
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting

```c
char p[1] = {0};
f();
print(p[0]);
```

constant prop.

```c
char p[1] = {0};
f();
print(0);
```
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting

```c
char p[1] = {0};
f();
print(p[0]);
```

constant prop.
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting

```c
char p[1] = {0};
f();
*(char*)(0x100)=1;
print(p[0]);
```

```c
constant
prop.
```

```c
char p[1] = {0};
f();
print(0);
```
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting
Problems with Full Provenance

Anyone can modify other’s local variables by
1. Guessing their addresses &
2. Acquiring full provenance via casting

```c
char p[1] = {0};
f();
print(p[0]);
(char*)(0x100) = 1;
```

Guessing

```
(char*) (0x100) = 1;
```

Full Provenance

constant prop.

```
char p[1] = {0};
f();
print(0);
```
Our Solution

Basic Idea
Exploit Nondeterministic Allocation

char p[1] = {0};
f();
*(char*)(0x100) = 1;
print(p[0]);

char p[1] = {0};
f();
print(0);
Our Solution

Basic Idea
Exploit Nondeterministic Allocation

char p[1] = {0};
f();
print(p[0]);

constant prop.

char p[1] = {0};
f();
print(0);
Our Solution

Basic Idea
Exploit Nondeterministic Allocation

char p[1] = {0};
f();
print(p[0]);

(p,0x100) (p,0x200)
(char*)=1;

(char*)=(0x100)

const prop.

char p[1] = {0};
f();
print(0);
Our Solution

Basic Idea
Exploit Nondeterministic Allocation

char p[1] = {0};
f();
*(char*)(0x100) = 1;
print(p[0]);

UB in Exec. 2:
no object at 0x100

constant prop.
More Formally, Twin Allocation

Memory: 0x100 0x200

```c
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```
More Formally, Twin Allocation

```c
char p[1] = {0};
*(char*)0x100 = 1;
print(p[0]);
```
More Formally, Twin Allocation

Memory:

 Exec. 1

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```
More Formally, Twin Allocation

Memory:

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x200</th>
</tr>
</thead>
</table>

```c
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```
More Formally, Twin Allocation

```
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```

Memory:

- Exec. 1: 
  - (p, 0x100)

- Exec. 2: 
  - (p, 0x200)

UB in Exec. 2: inaccessible at 0x100
More Formally, Twin Allocation

N.B.
This argument works only for unobserved addresses

UB in Exec. 2: inaccessible at 0x100

char p[1] = {0};
*(char*)(0x100) = 1;
printf(p[0]);

(p,0x100) (p,0x200)
Example with Observed Address

```c
char p[1] = {0};
*(char*)(0x100) = 1;
print(p[0]);
```
Example with Observed Address

```c
char p[1] = {0};
if (p == 0x100){
    *(char*)(0x100) = 1;
}
print(p[0]);
```
char p[1] = {0};
if (p == 0x100){
    *(char*)(0x100) = 1;
}
print(p[0]);
char p[1] = {0};
if (p == 0x100){
    *(char*)(0x100) = 1;
}
print(p[0]);
Example with Observed Address

```c
char p[1] = {0};
if (p == 0x100)
    *(char*)(0x100) = 1;
print(p[0]);
```

No UB in Exec. 2
Consistent with common compilers’ assumption: Observed variables can be modified by others

```c
char p[1] = {0};
if (p == 0x100) {
    *(char*)(0x100) = 1;
}
p[0];
```

Example with Observed Address

Exec. 1

Exec. 2

(p,0x100) (p,0x200)

No UB in Exec. 2
Miscompilation Revisited

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

**Constant Propagation**

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```
```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```
Can Access q[0] due to Full Prov.

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)(int)(p+1) = 10;
  print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(char*)iq = 10;
  print(q[0]);
}
```

```
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
  *(p+1) = 10;
  print(q[0]);
}
```

**Miscompilation Revisited**

int. eq. prop.

cast elim.

constant prop.
Miscompilation Revisited

Can Access q[0] due to Full Prov.

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(char*)(int)(p+1) = 10;
    print(q[0]);
}
```

Cannot Access q[0] due to Prov. p

```c
char p[1], q[1] = {0};
int ip = (int)(p+1);
int iq = (int)q;
if (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}
```
Miscompilation Revisited

If (iq == ip) {
    *(p+1) = 10;
    print(q[0]);
}

Can Access q[0] due to Full Prov.

Ptr → Int → Ptr Cast Elimination Is Unsound:
A Potential Performance Issue

due to Prov. p

constant prop.
Solution to the Cast Elim. Problem

Reducing # of Int ↔ Ptr Casts

• Most casts are introduced by compilers for convenience
• We recovered performance by reducing unnecessary casts
  - Int → Ptr: 95% removed
  - Ptr → Int: 75% removed
Solution to the Cast Elim. Problem

The paper includes more details & a formal specification
Implementation & Evaluation

- We fixed LLVM 6.0 to be sound in our memory model
- We had to change only 1.7K LOC in total

Benchmark Results
- SPEC CPU2017: <0.1% avg, <0.5% max slowdown
- LLVM Nightly Tests: <0.1% avg, <3% max slowdown

- We verified key properties of our memory model in Coq
Conclusion

• We develop a memory model for IR which supports both low-level code & high-level optimizations

• We use full provenance & twin allocation to reconcile them

• Applying our model to LLVM has little impact on performance