A Formal C Memory Model Supporting Integer-Pointer Casts

Jeehoon Kang (Seoul National University)

Chung-Kil Hur (Seoul National University)

William Mansky (UPenn)

Dmitri Garbuzov (UPenn)

Steve Zdancewic (UPenn)

Viktor Vafeiadis (MPI-SWS)

@INRIA Paris

Motivation

- **Integer-pointer cast is an important feature of C.**
  - used in Linux kernel, Java HotSpot VM
- **Pointers being integers invalidates optimizations.**
  - e.g. constant propagation
- **Want to support integer-pointer casts & optimizations**

Integer-Pointer Casts:
Importance in Practice

- **Example 1: Pointers as hash keys**

```
void hash_put(void* key, Data value);
Data hash_get(void* key);
```

- **Example 2: Pointer compression in Java HotSpot VM**

```
int32_t compress(void*);    // 64bit -> 32bit
void* decompress(int32_t); // 32bit -> 64bit
```

Identifying Pointers with Integers: Invalidates Constant Propagation

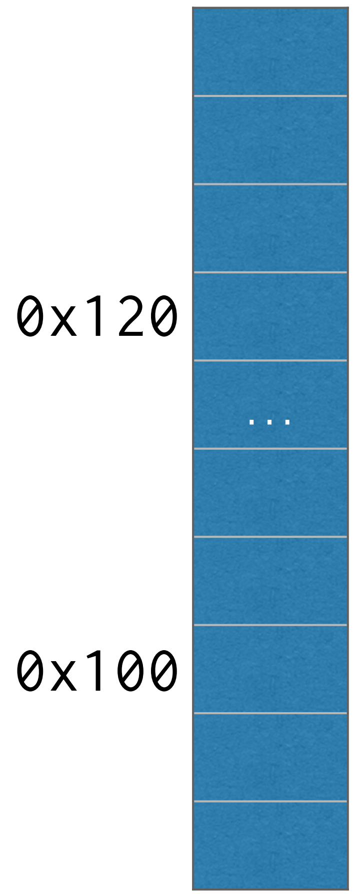
- **Anyone can access any address.**

```
extern void g();




char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

Identifying Pointers with Integers:
Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

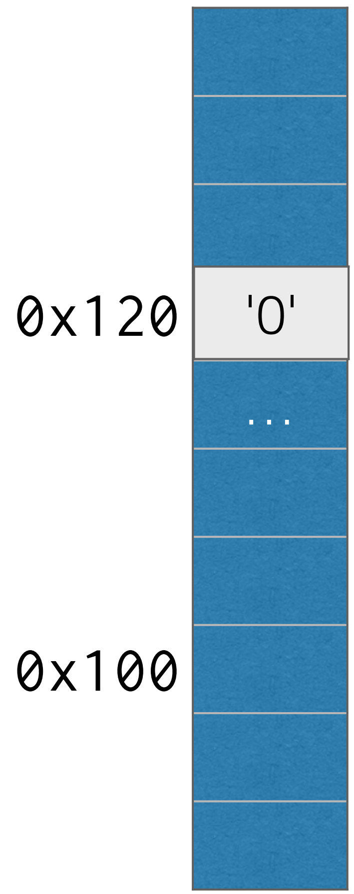


expr	value

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

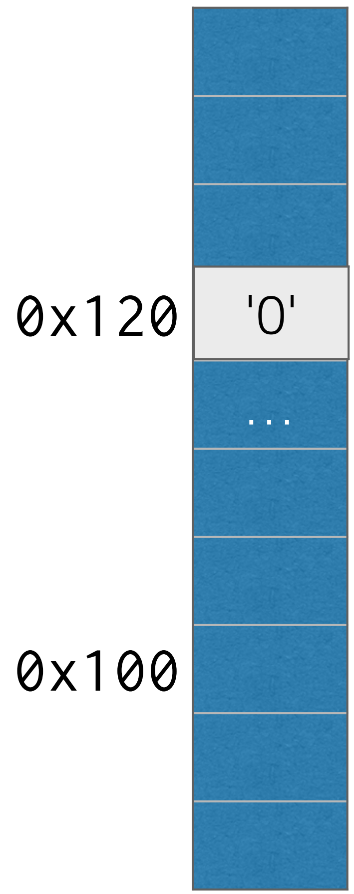


expr	value
&a	0x120

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

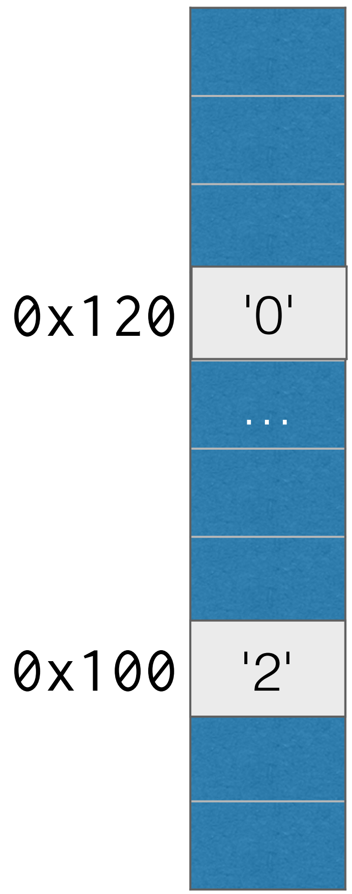


expr	value
&a	0x120

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

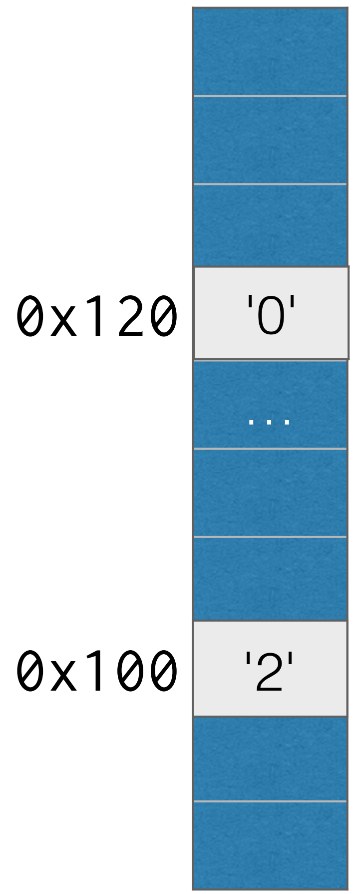


expr	value
&a	0x120
&b	0x100

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

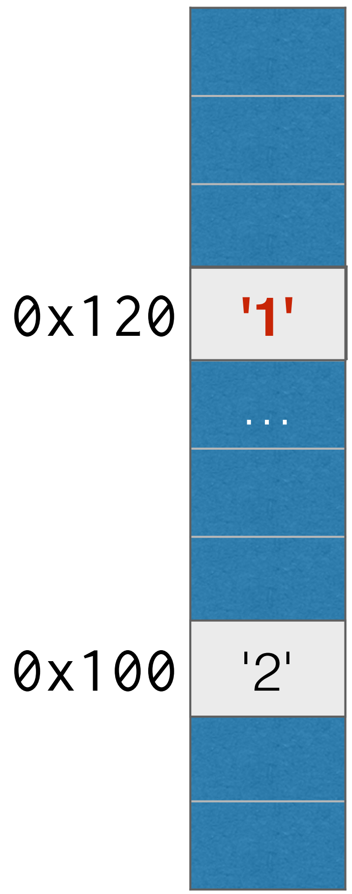


expr	value
&a	0x120
&b	0x100
&b+0x20	0x120

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```

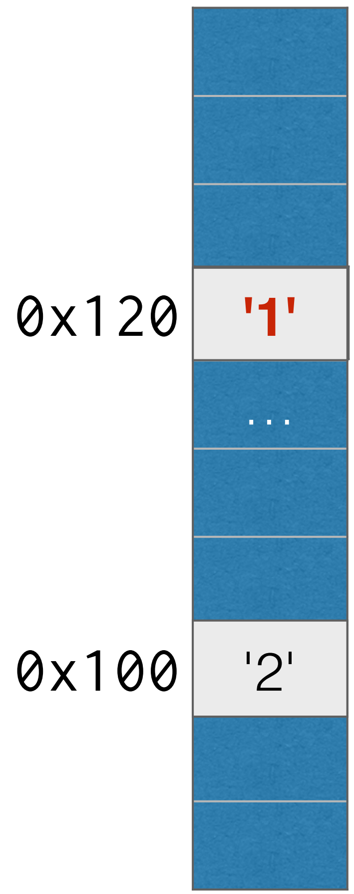


expr	value
&a	0x120
&b	0x100
&b+0x20	0x120

Identifying Pointers with Integers: Invalidates Constant Propagation

- **Anyone can access any address.**

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> return '0'
}
```



expr	value
&a	0x120
&b	0x100
&b+0x20	0x120

Goal of Memory Model

- To validate common optimizations
  by disallowing problematic memory accesses

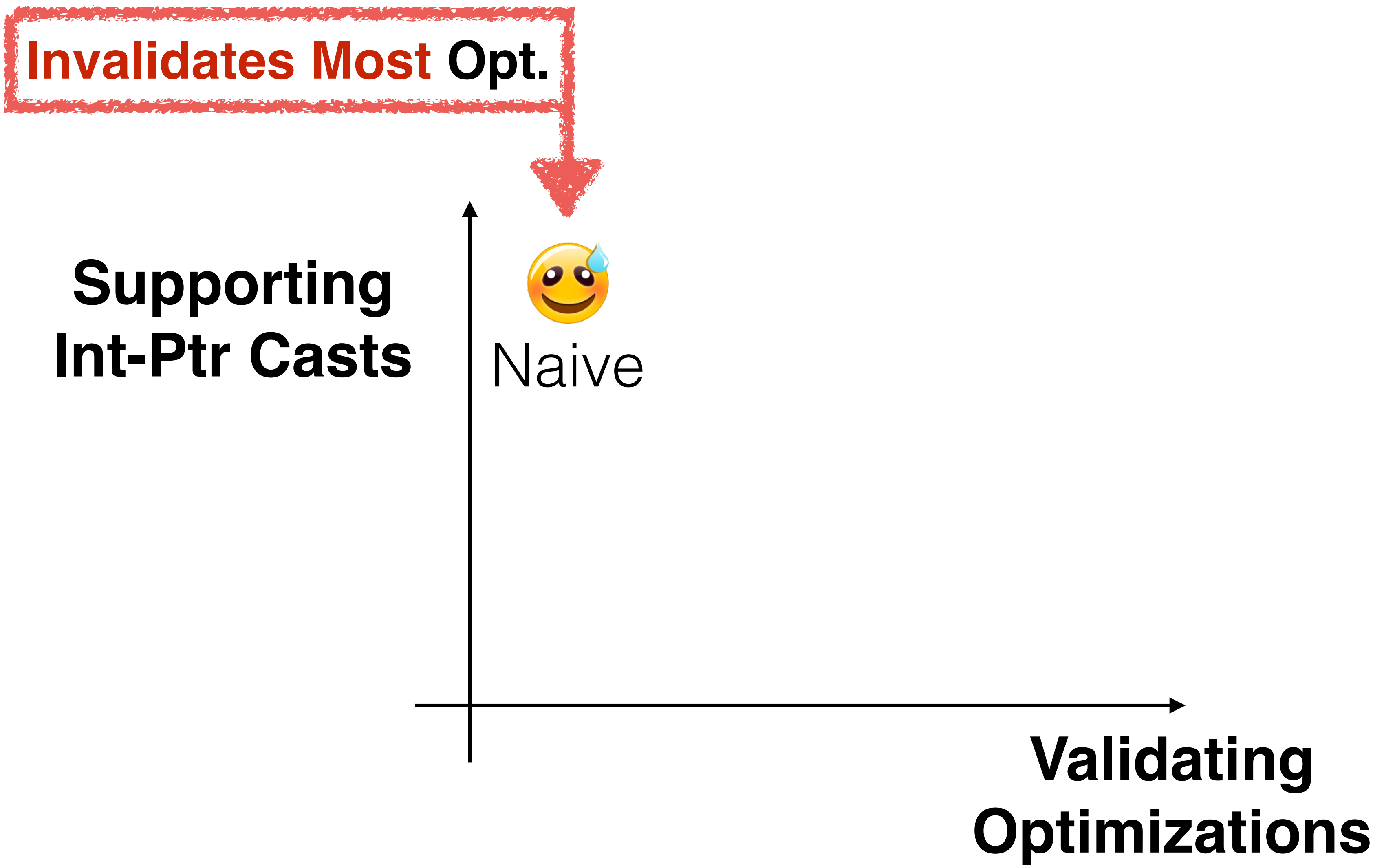
- To allow integer-pointer casts

Outline

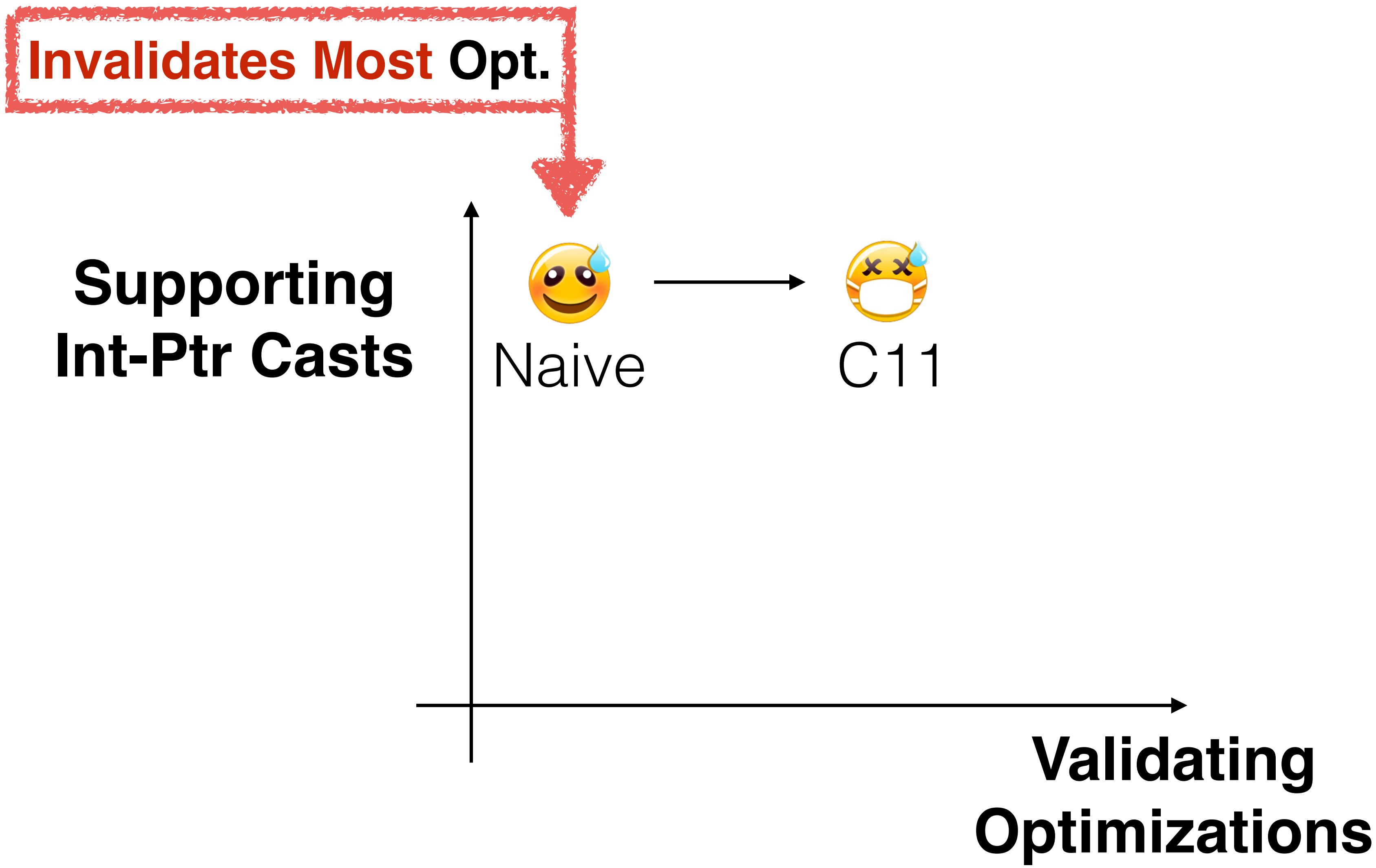
Supporting Int-Ptr Casts

Validating Optimizations

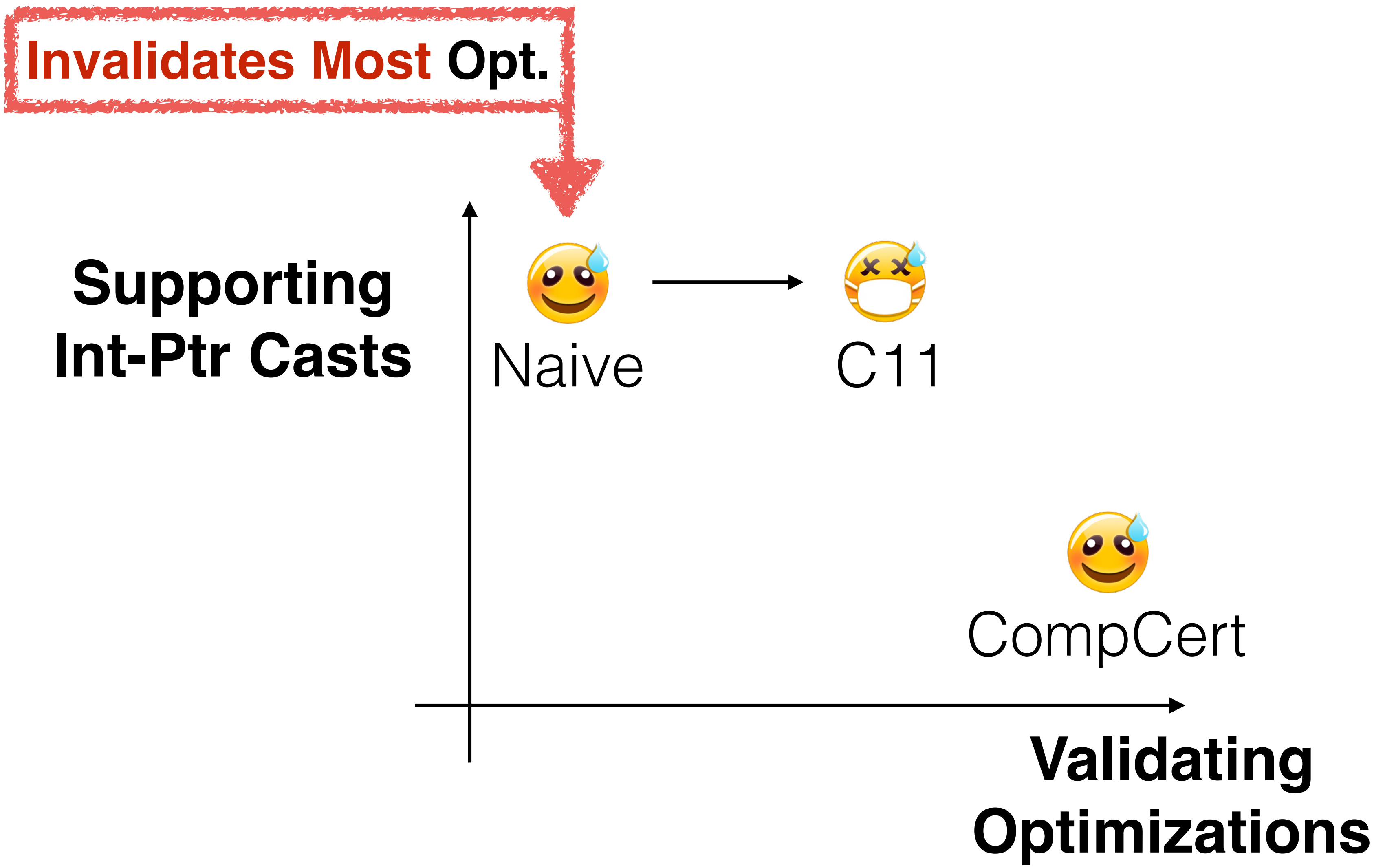
Outline


Invalidates Most Opt.
Supporting Int-Ptr Casts
Naive
Validating Optimizations

Outline

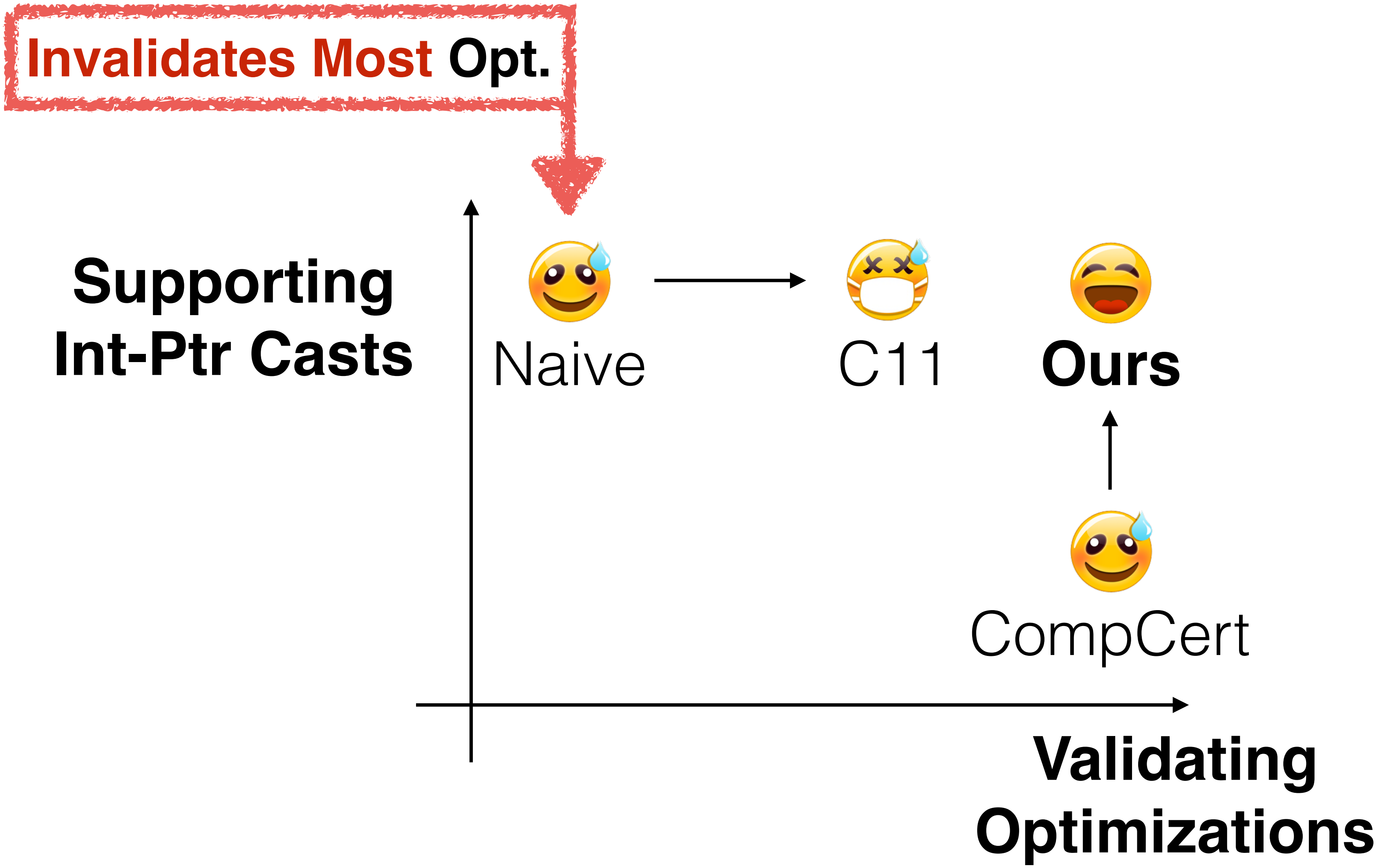

Invalidates Most Opt.
Supporting
Int-Ptr Casts
Naive
C11
Validating
Optimizations

Outline

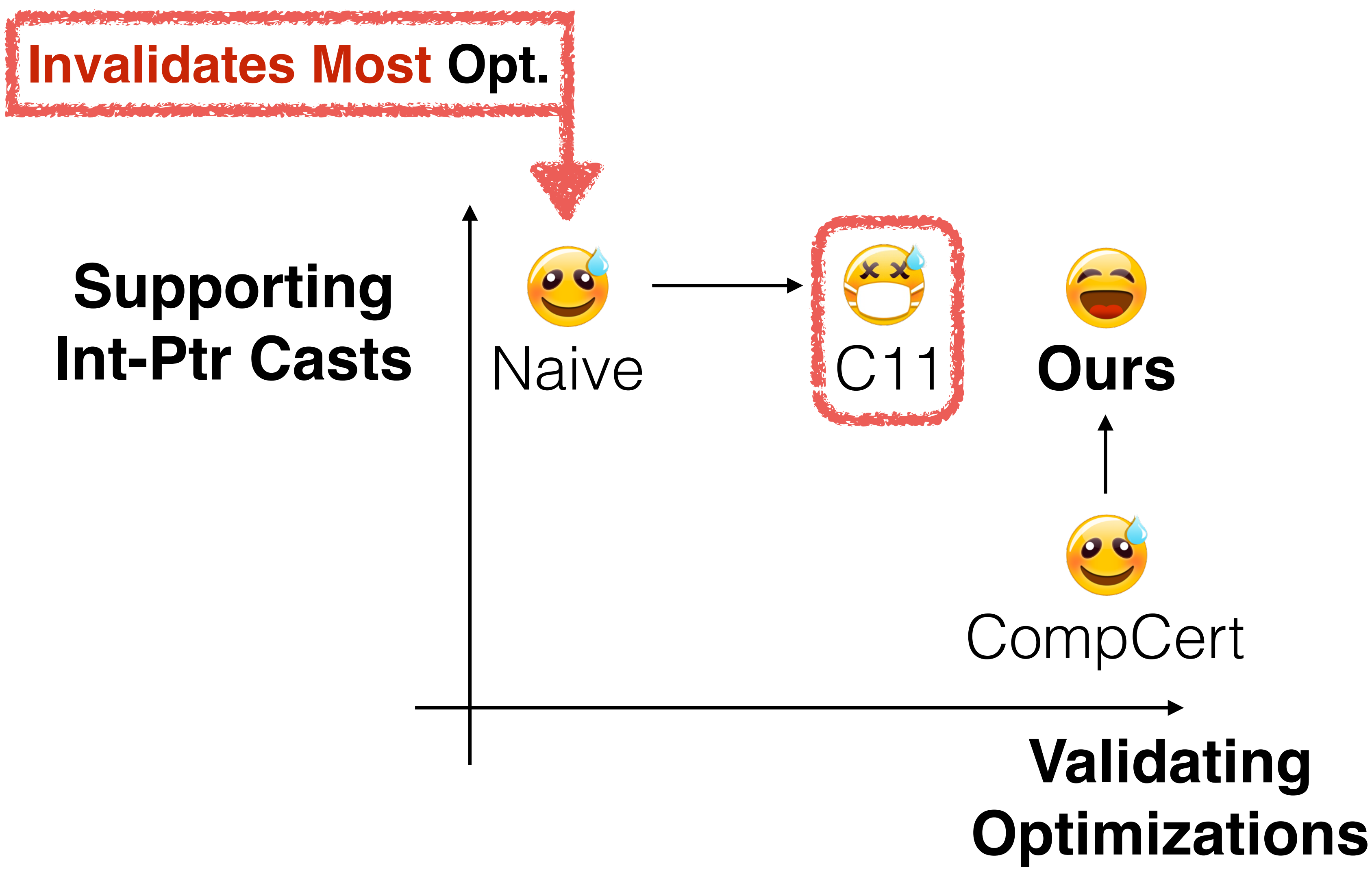


Invalidates Most Opt.
Supporting
Int-Ptr Casts
Naive
C11
CompCert
Validating
Optimizations

Outline


Invalidates Most Opt.
Supporting
Int-Ptr Casts
Naive
C11
Ours
CompCert
Validating
Optimizations

Outline



Invalidates Most Opt.
Supporting Int-Ptr Casts
Naive
C11
Ours
CompCert
Validating Optimizations

C11 Model:
High-Level Idea

- **Integers & pointers are tagged with permission.**

Integers		**Pointers**
`0x120`	⟷	`0x120`
`0x100`	⟷	`0x100`

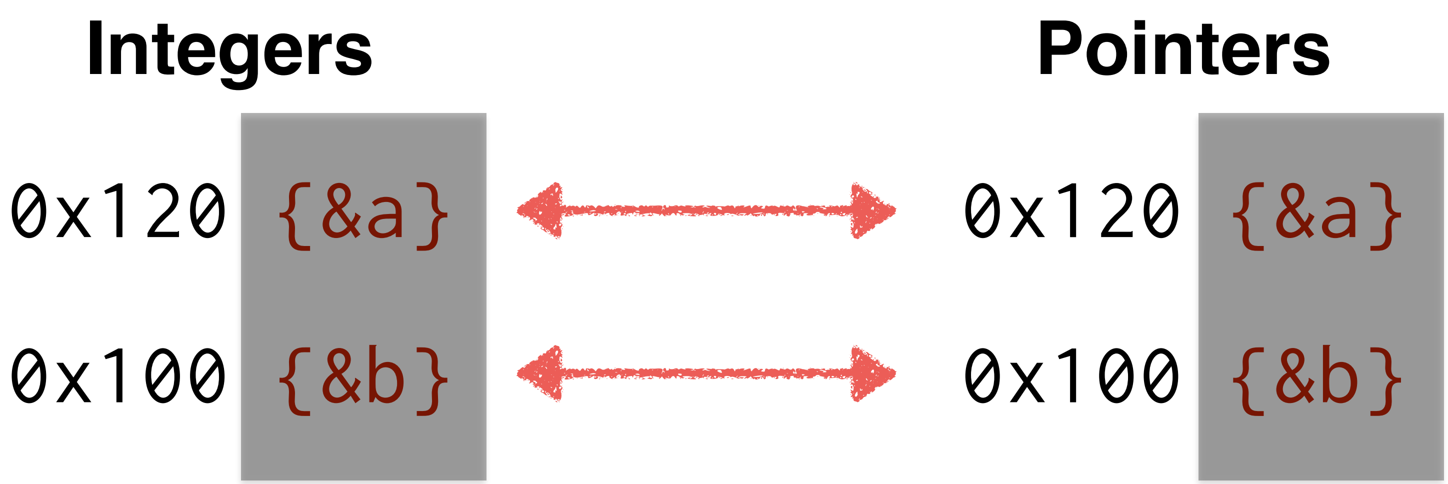
C11 Model:
High-Level Idea

- **Integers & pointers are tagged with permission.**

Integers		**Pointers**
`0x120 {&a}`	⟷	`0x120 {&a}`
`0x100 {&b}`	⟷	`0x100 {&b}`

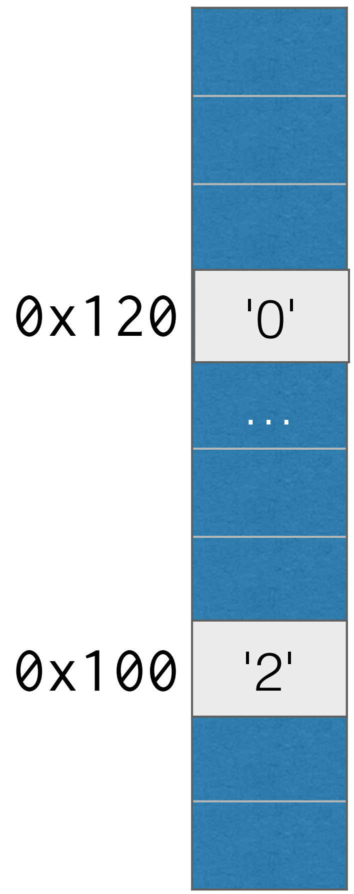
C11 Model:
High-Level Idea

- **Integers & pointers are tagged with permission.**



C11 Model:
Protection by Permission

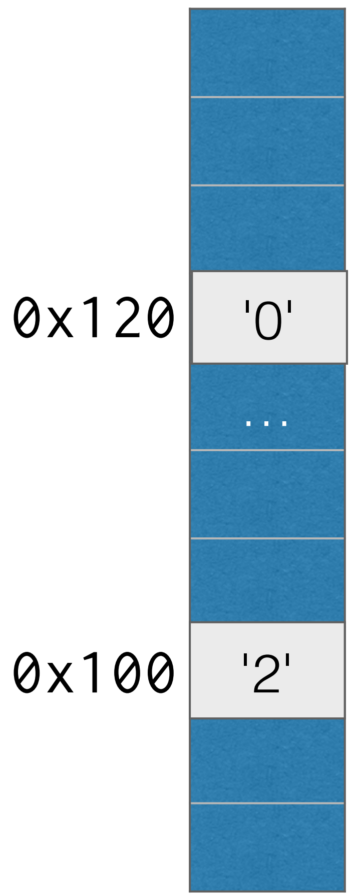
```c
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



expr	value {perm.}
&a	0x120
&b	0x100
&b+0x20	0x120

C11 Model:
Protection by Permission

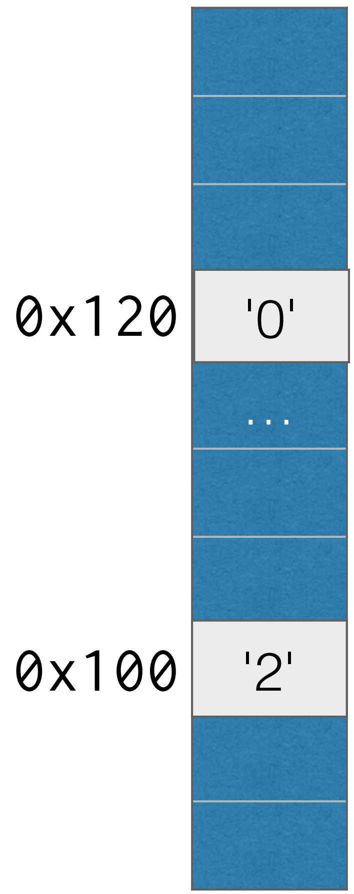
```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



expr	value {perm.}
&a	0x120 {&a}
&b	0x100
&b+0x20	0x120

C11 Model:
Protection by Permission

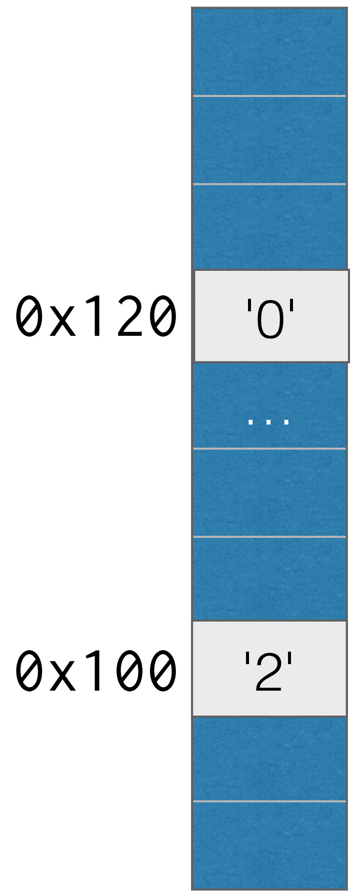
```c
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



expr	value {perm.}
&a	0x120 {&a}
&b	0x100 {&b}
&b+0x20	0x120

C11 Model:
Protection by Permission

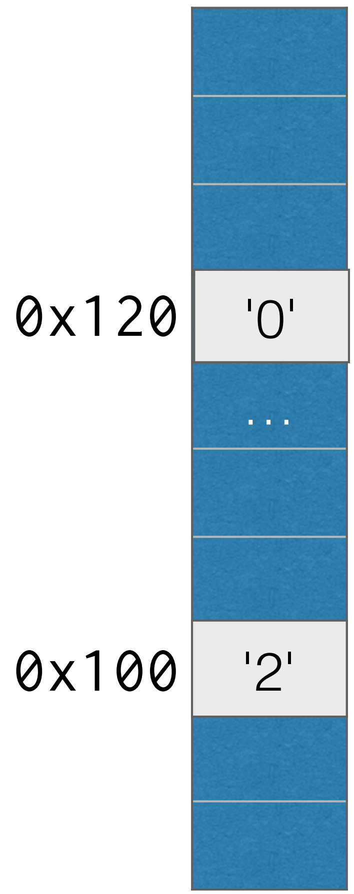
```c
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



expr	value {perm.}
&a	0x120 {&a}
&b	0x100 {&b}
&b+0x20	0x120 {&b}

C11 Model:
Protection by Permission

```
void g() {
 char b = '2';
 char* p = &b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



expr	value {perm.}
&a	0x120 {&a}
&b	0x100 {&b}
&b+0x20	0x120 {&b}

Cannot Access a

C11 Model's Problems (1/2): Too Complex Semantics

- **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.
- **Operations need to properly calculate permission.**

```
int y = x - x; // -> int y = 0;
```

expr	value
`x`	`0x100 {&a}`
`x - x`	`0x000 {? } -> 0x000 {}`

C11 Model's Problems (1/2):
Too Complex Semantics

- **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.
- **Operations need to properly calculate permission.**

```
int y = x - x; // -> int y = 0;
```

expr	value
`x`	`0x100 {&a}`
`x - x`	`0x000 {&a} -> 0x000 {}`

C11 Model's Problems (1/2):
Too Complex Semantics

- **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.
- **Operations need to properly calculate permission.**

```
int y = x - x; // -> int y = 0;
```

expr	value
`x`	`0x100 {&a}`
`x - x`	`0x000 {  } -> 0x000 {}`

C11 Model's Problems (1/2): Too Complex Semantics

- **Integers also need to carry permission.**
  Since integer-pointer casts should preserve permission.

- **Operations need to properly calculate permission.**

```
int y = x - x; // -> int y = 0;
```

expr	value
`x`	`0x100 {&a}`
`x - x`	`0x000 {  } -> 0x000 {}`
`2 * x`	`0x200 {? }`
`x XOR x`	`0x000 {? }`
…	…

C11 Model's Problems (2/2):
Invalidates Some Optimizations

- **A useful code motion is not allowed.**

```
int a, b;
…
if (a != b) {
  a = b;
}
```

```
int a, b;
…
if (a != b) {
}
a = b;
```

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **A useful code motion is not allowed.**

```
int a, b;
…
if (a != b) {
  a = b;
}
```

```
int a, b;
…
if (a != b) {
}
a = b;
```

expr	value
a	`0x100 {&x} -> 0x100 {}`
b	`0x100 {}`

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

Integer type for pointers

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2):
Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

DEAD CODE

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
    xi = i;
  }
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 0
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

DEAD CODE

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 0
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

Integer type for pointers

DEAD CODE

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 0
```

i | … {}

xi | … {}

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752

C11 Model's Problems (2/2): Invalidates Some Optimizations

- **We found a real GCC bug.**

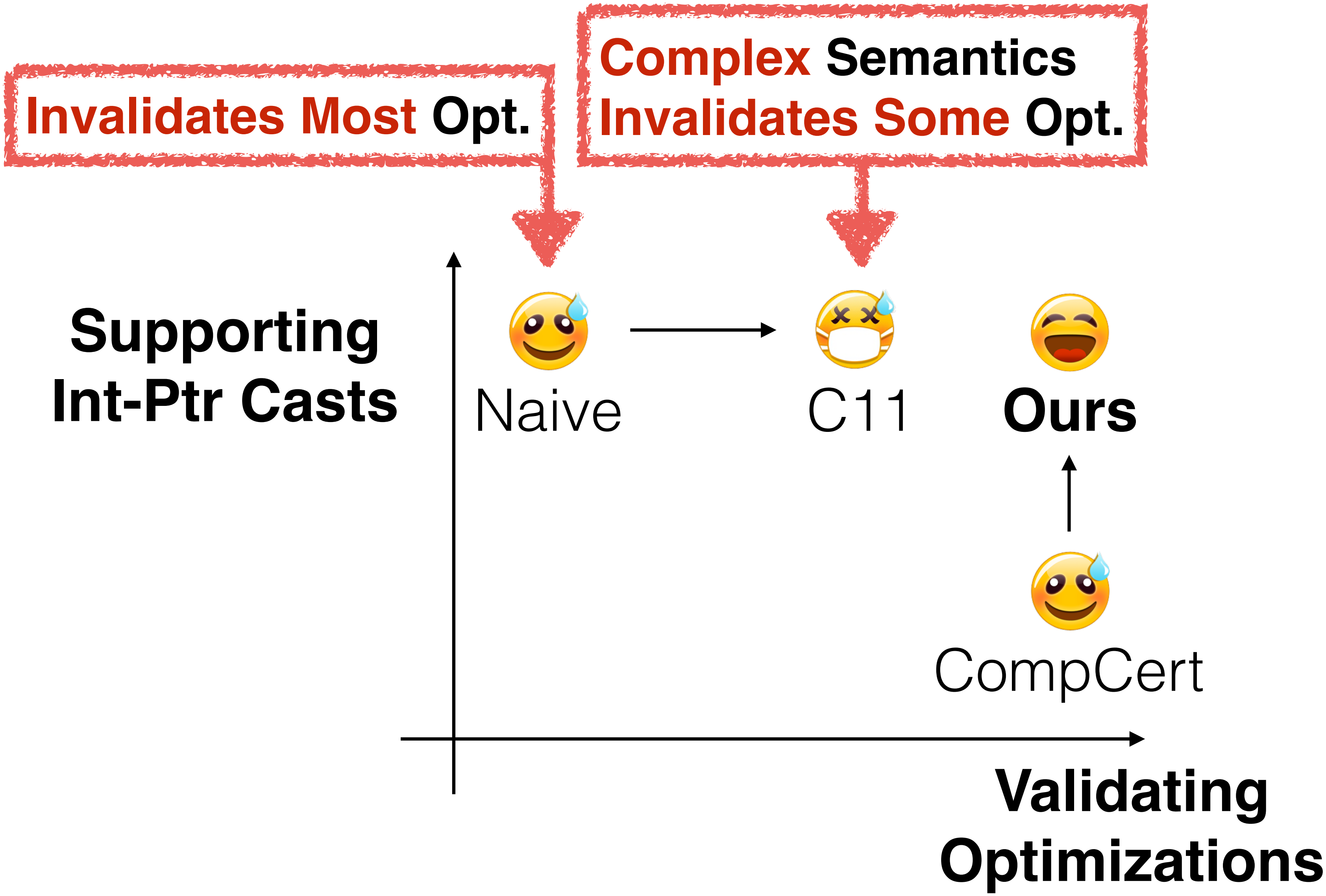
Integer type for pointers

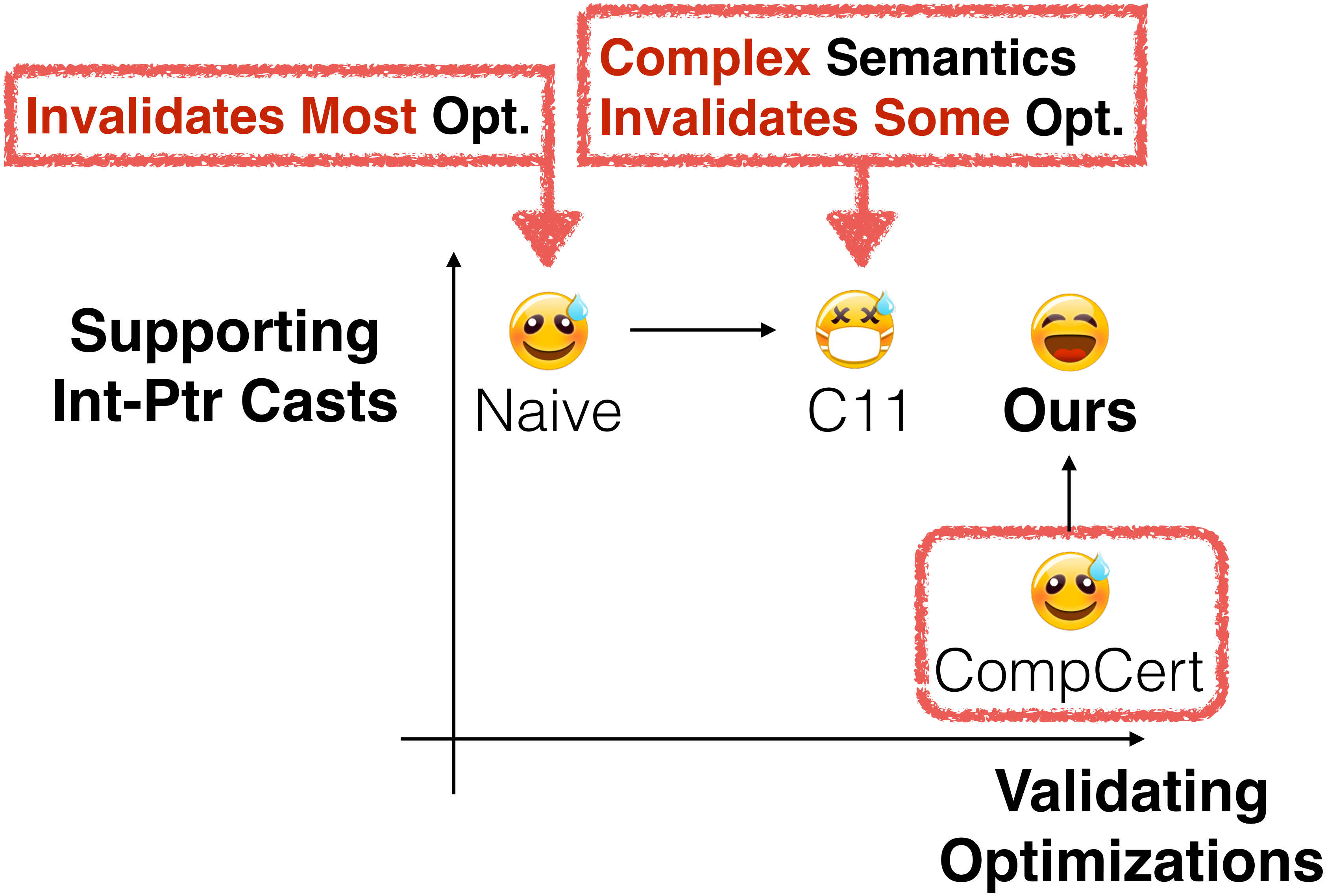
DEAD CODE

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", 0); } // constant propagation x -> 0
```

```
i  | ... {}
xi | ... {}
```

https://gcc.gnu.org/bugzilla/show_bug.cgi?id=65752


Invalidates Most Opt.
Complex Semantics
Invalidates Some Opt.
Supporting
Int-Ptr Casts
Naive
C11
Ours
CompCert
Validating
Optimizations


Invalidates Most Opt.
Complex Semantics
Invalidates Some Opt.
Supporting
Int-Ptr Casts
Naive
C11
Ours
CompCert
Validating
Optimizations

CompCert Model:
High-Level Idea

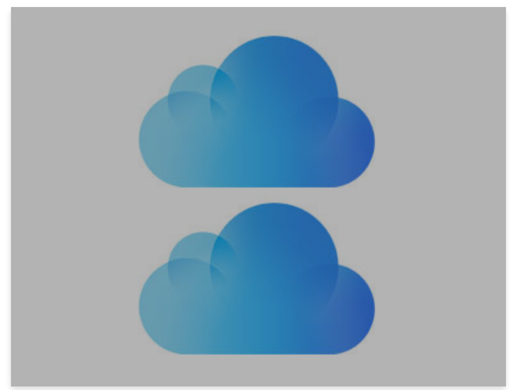
- **Pointers are different from integers.**

Integers

`0x120`

`0x100`

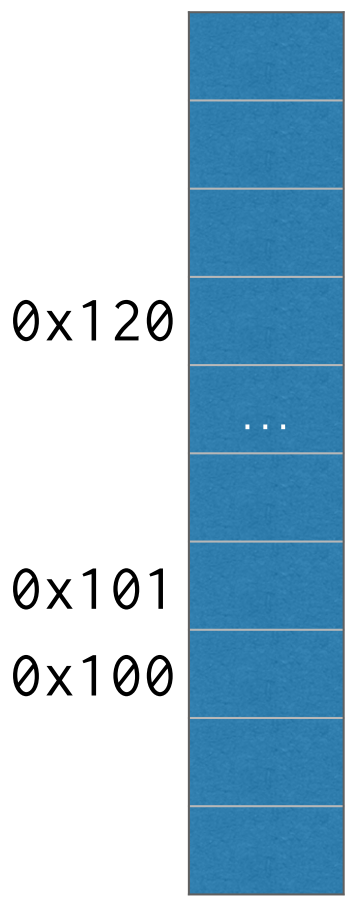
Pointers



CompCert Model:
Protection by Logical Blocks

```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```

Naive

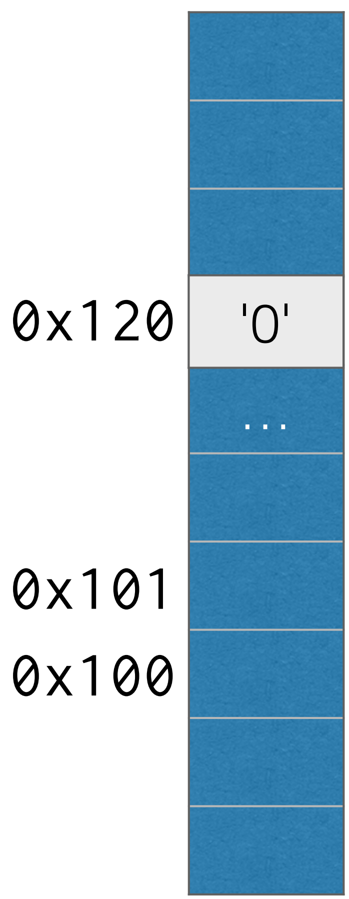


CompCert

CompCert Model:
Protection by Logical Blocks

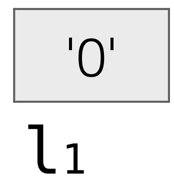
```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```

Naive



&a	0x120

CompCert



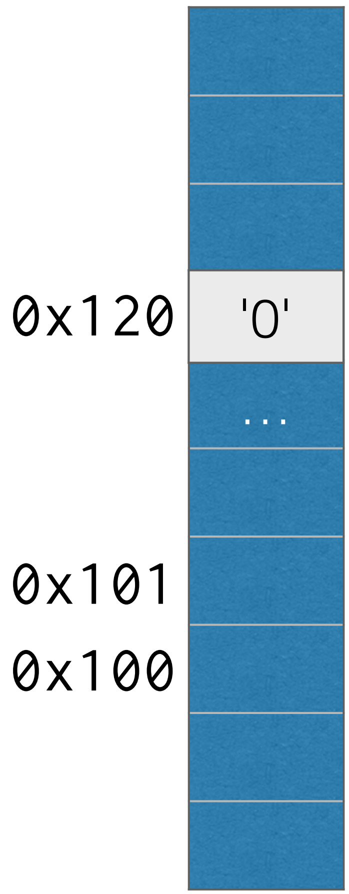
&a	(l_1,0)

CompCert Model:
Protection by Logical Blocks

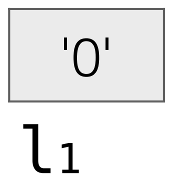
```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```

&a	0x120

Naive



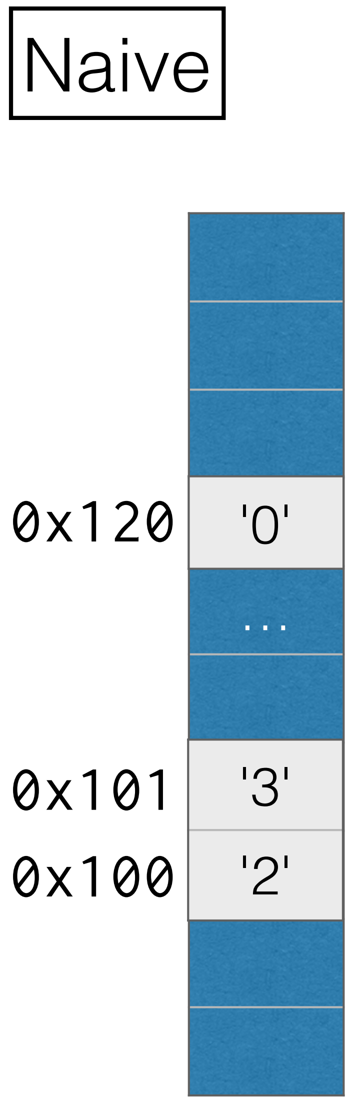
CompCert



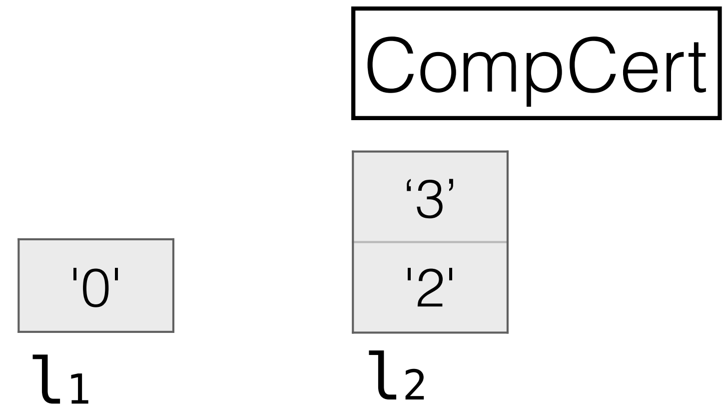
&a	$(l_1, 0)$

CompCert Model:
Protection by Logical Blocks

```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



&a	0x120
b	0x100



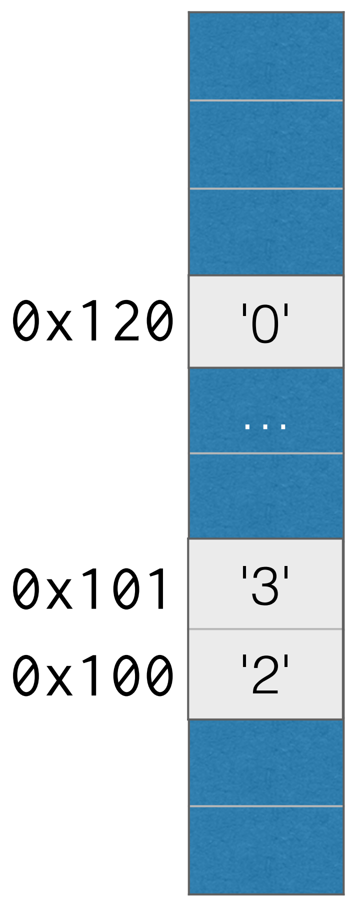
&a	$(l_1,0)$
b	$(l_2,0)$

CompCert Model:
Protection by Logical Blocks

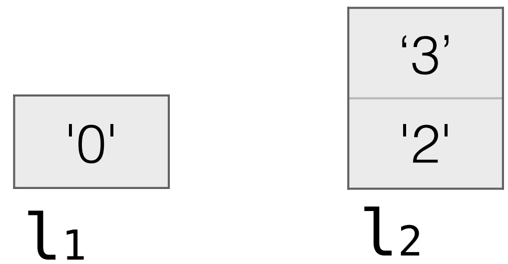
```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```

&a	0x120
b	0x100
b+0x20	0x120

Naive



CompCert



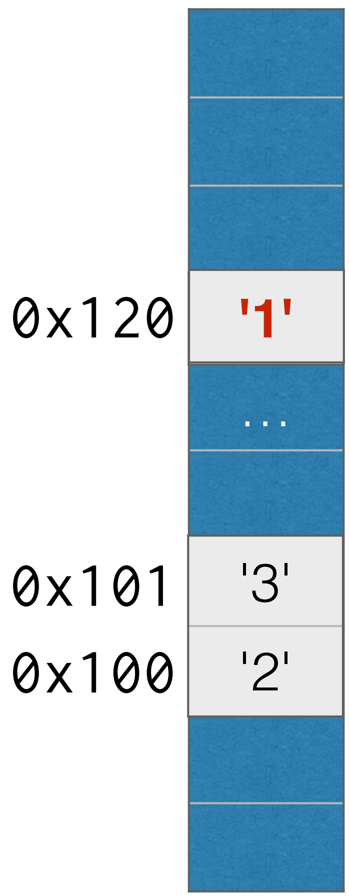
&a	$(l_1, 0)$
b	$(l_2, 0)$
b+0x20	$(l_2, 0x20)$

CompCert Model:
Protection by Logical Blocks

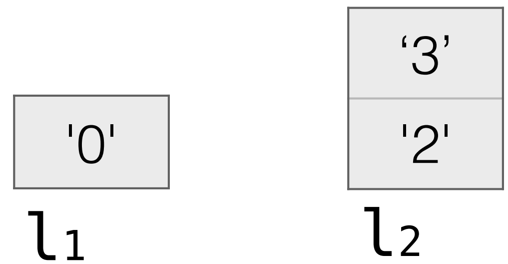
```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```

&a	0x120
b	0x100
b+0x20	0x120

Naive

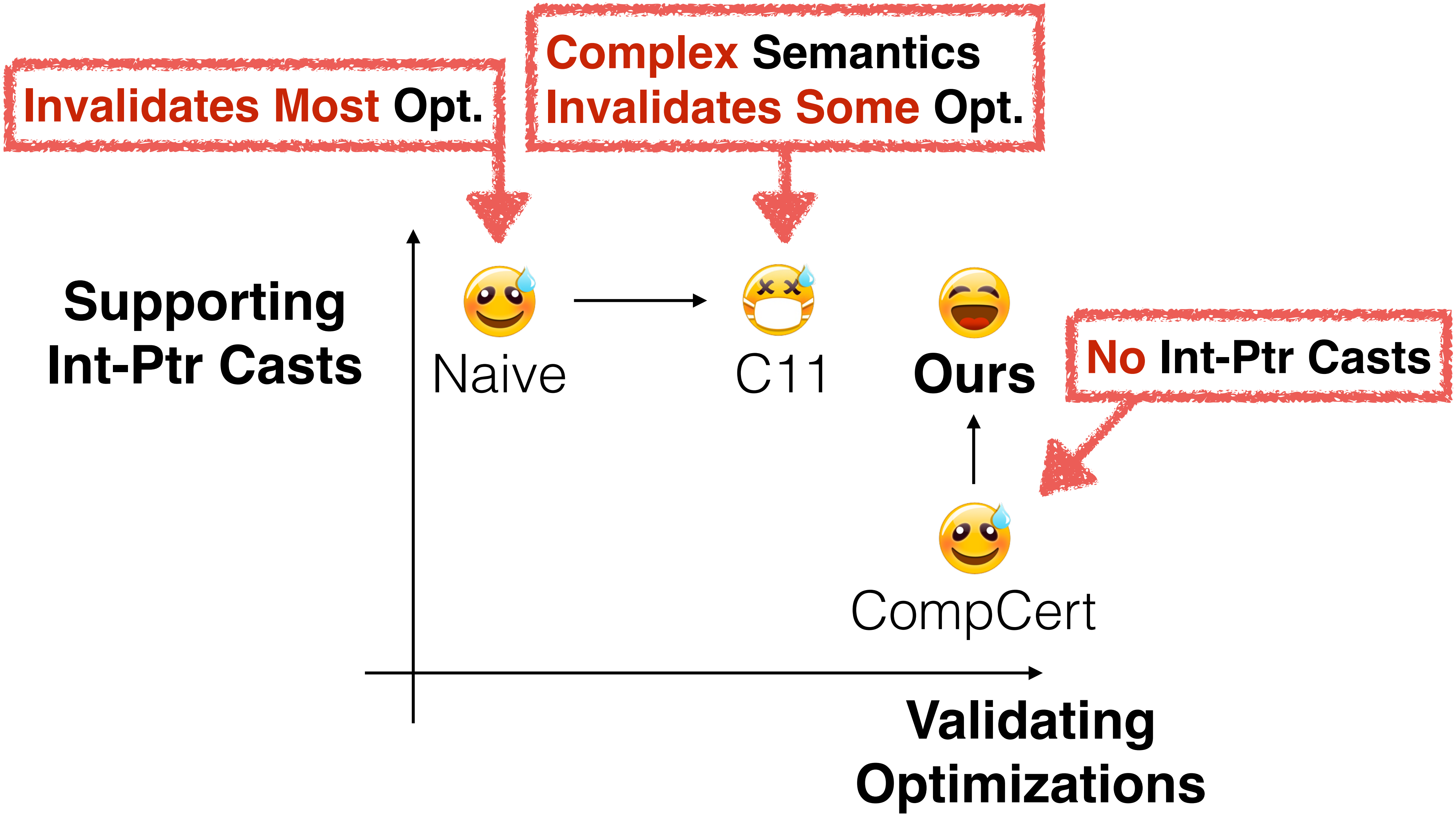


CompCert

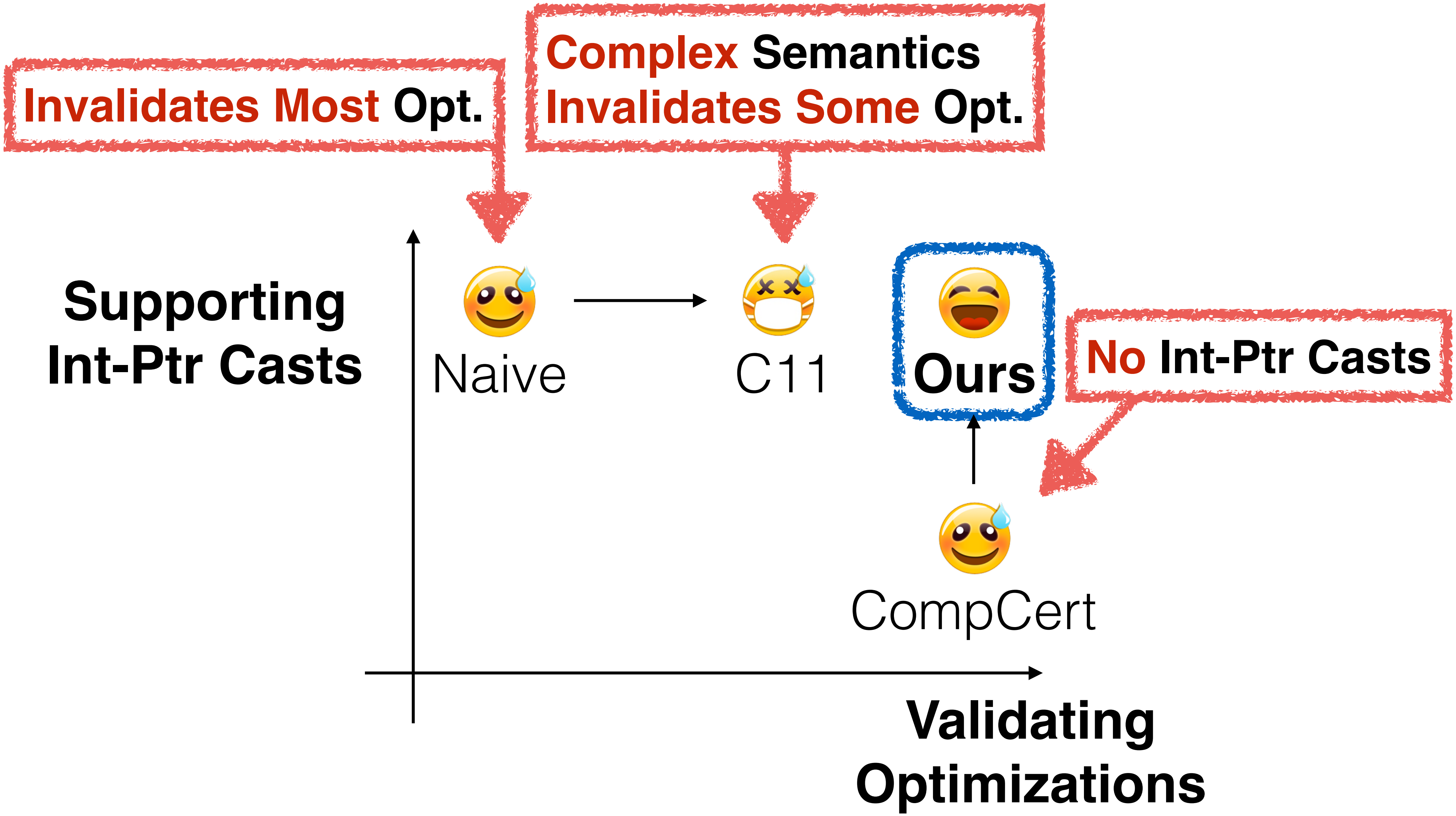


Cannot Access a

&a	$(l_1, 0)$
b	$(l_2, 0)$
b+0x20	$(l_2, 0x20)$



Invalidates Most Opt.
Complex Semantics
Invalidates Some Opt.
Supporting
Int-Ptr Casts
Naive
C11
Ours
No Int-Ptr Casts
CompCert
Validating
Optimizations



Invalidates Most Opt.
Complex Semantics
Invalidates Some Opt.
Supporting Int-Ptr Casts
Naive
C11
Ours
No Int-Ptr Casts
CompCert
Validating Optimizations

Our Model:
High-Level Idea

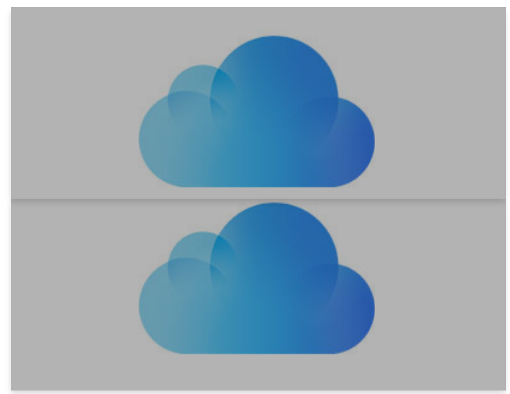
- **Pointers become integers only when casted.**

Integers

`0x120`

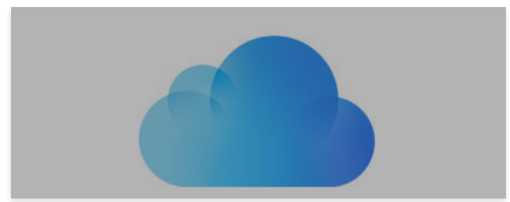
`0x100`

Pointers



Our Model:
High-Level Idea

- **Pointers become integers only when casted.**

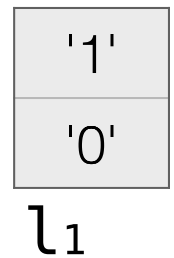
Integers		**Pointers**
		
`0x120`		
`0x100`		`0x100`

Our Model:
Realizes at Casting to Integer

```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```

Our Model:
Realizes at Casting to Integer

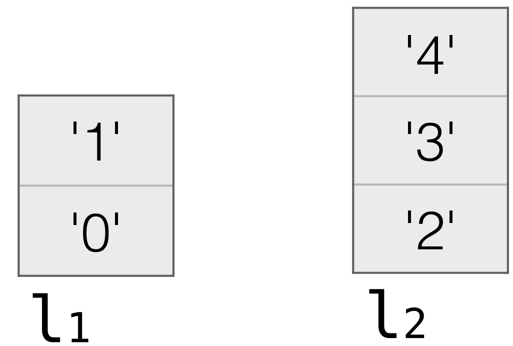
```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



a	$(l_1, 0)$

Our Model:
Realizes at Casting to Integer

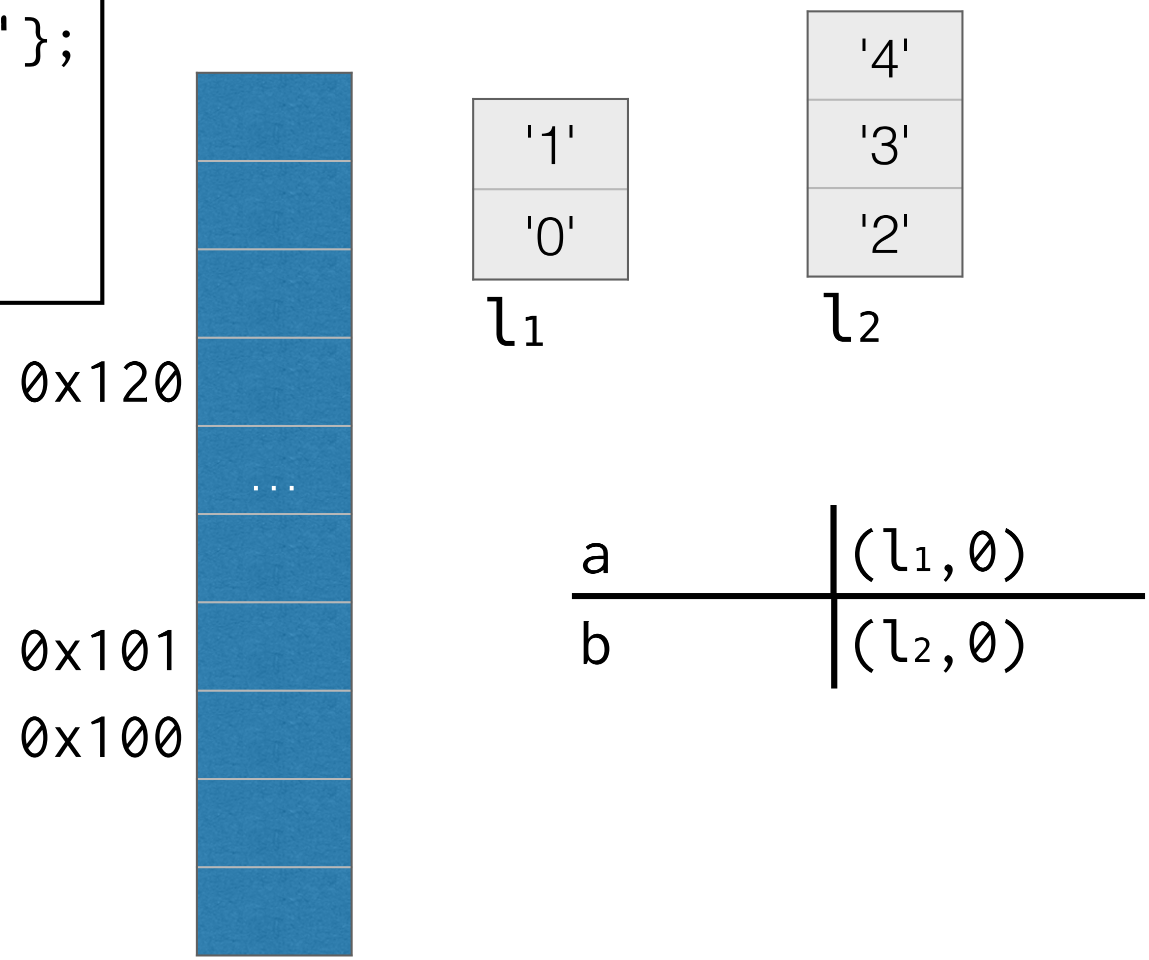
```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



a	$(l_1, 0)$
b	$(l_2, 0)$

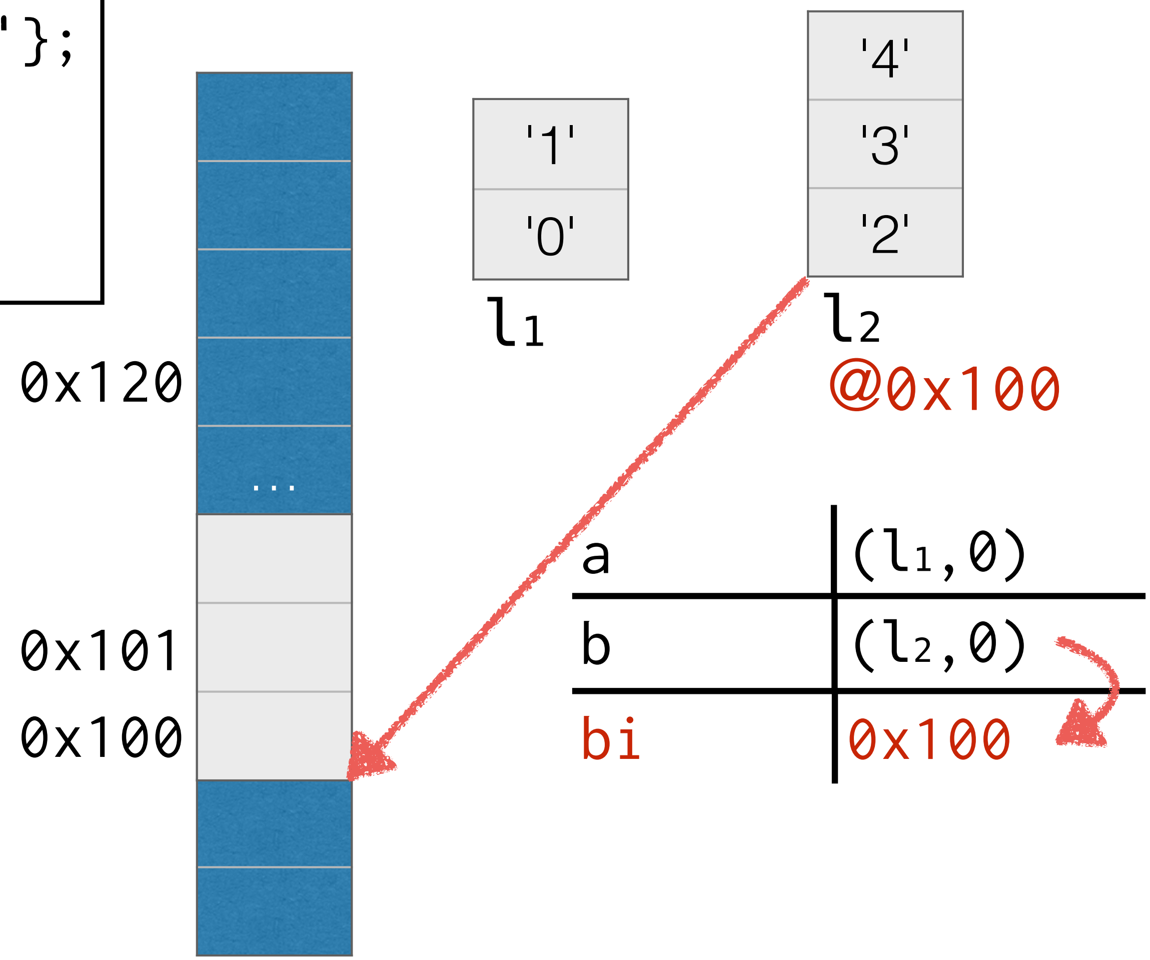
Our Model:
Realizes at Casting to Integer

```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



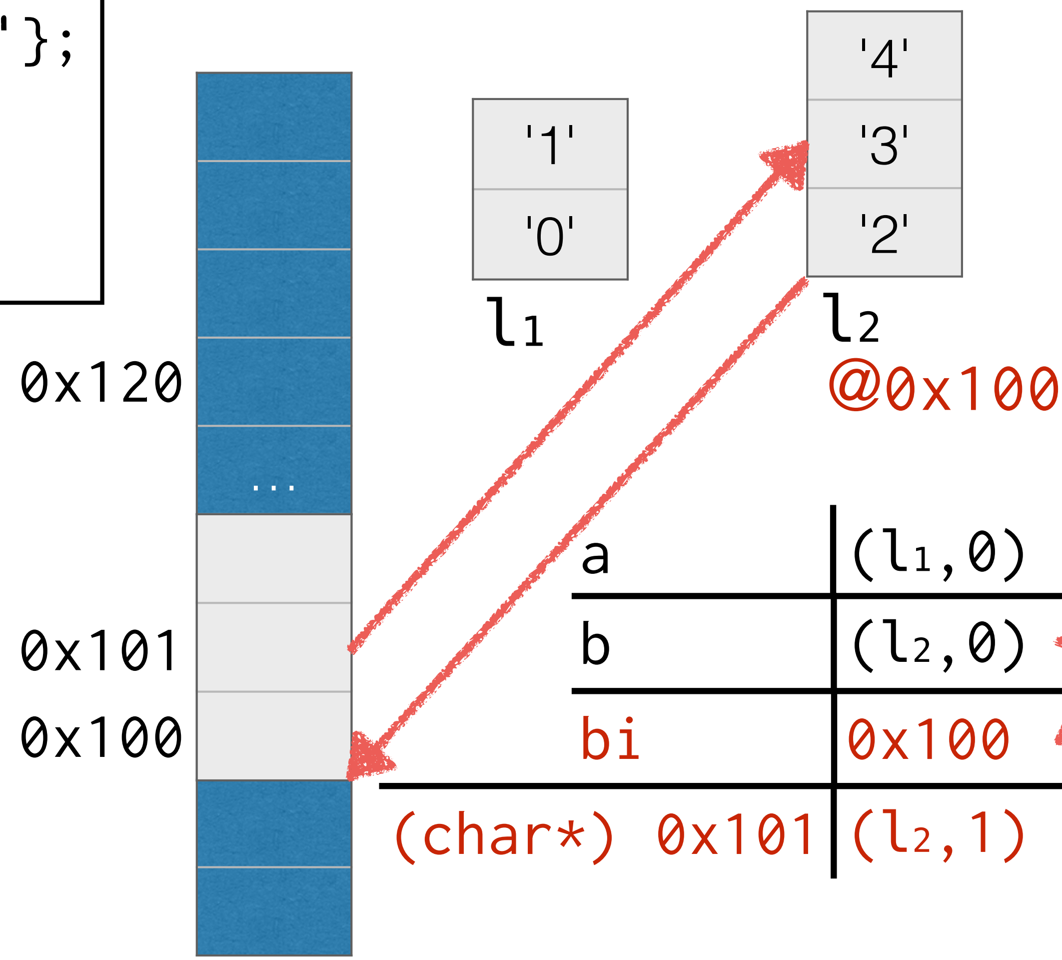
Our Model:
Realizes at Casting to Integer

```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



Our Model:
Realizes at Casting to Integer

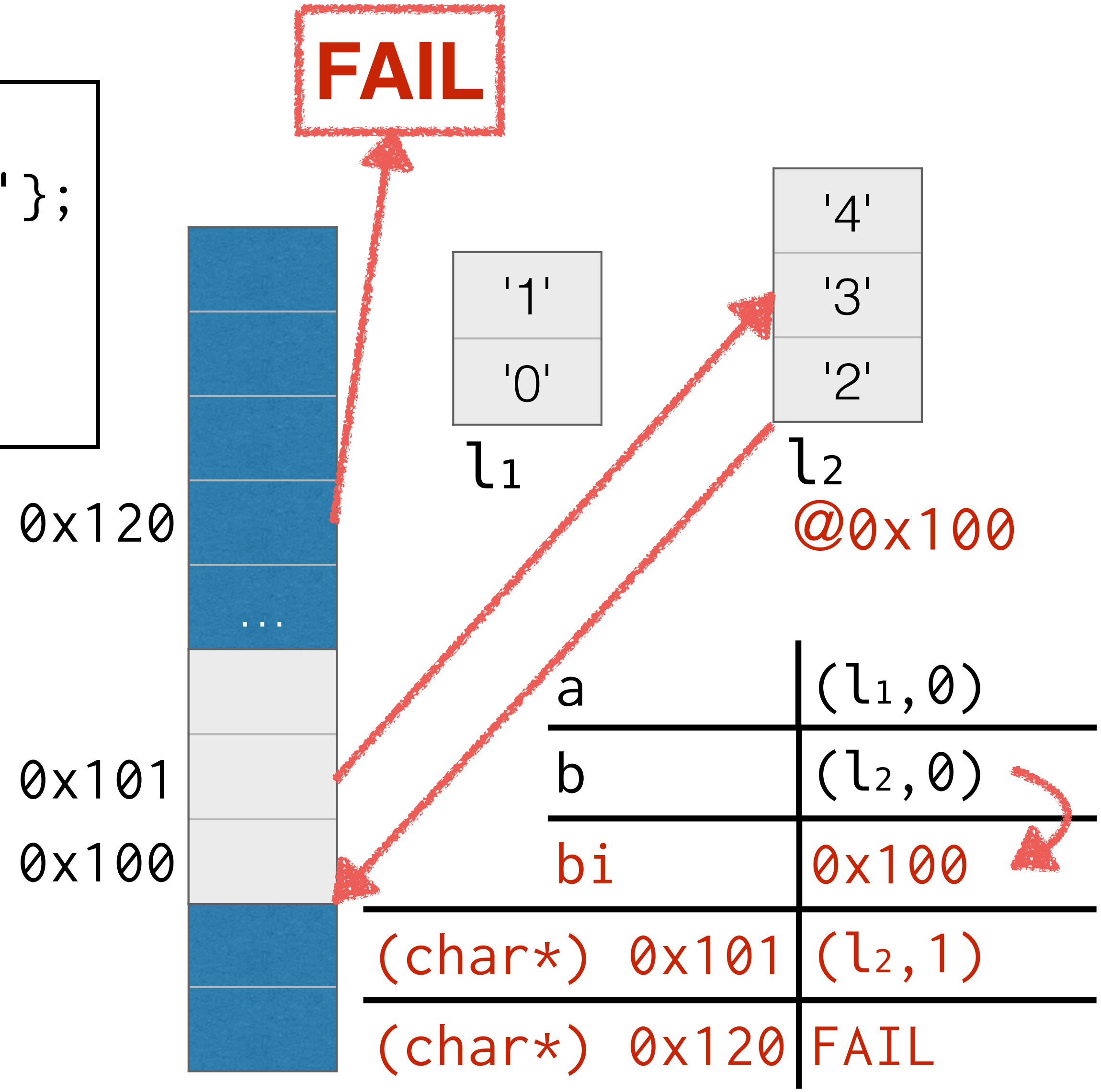
```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



a	(l1,0)
b	(l2,0)
bi	0x100
(char*) 0x101	(l2,1)

Our Model:
Realizes at Casting to Integer

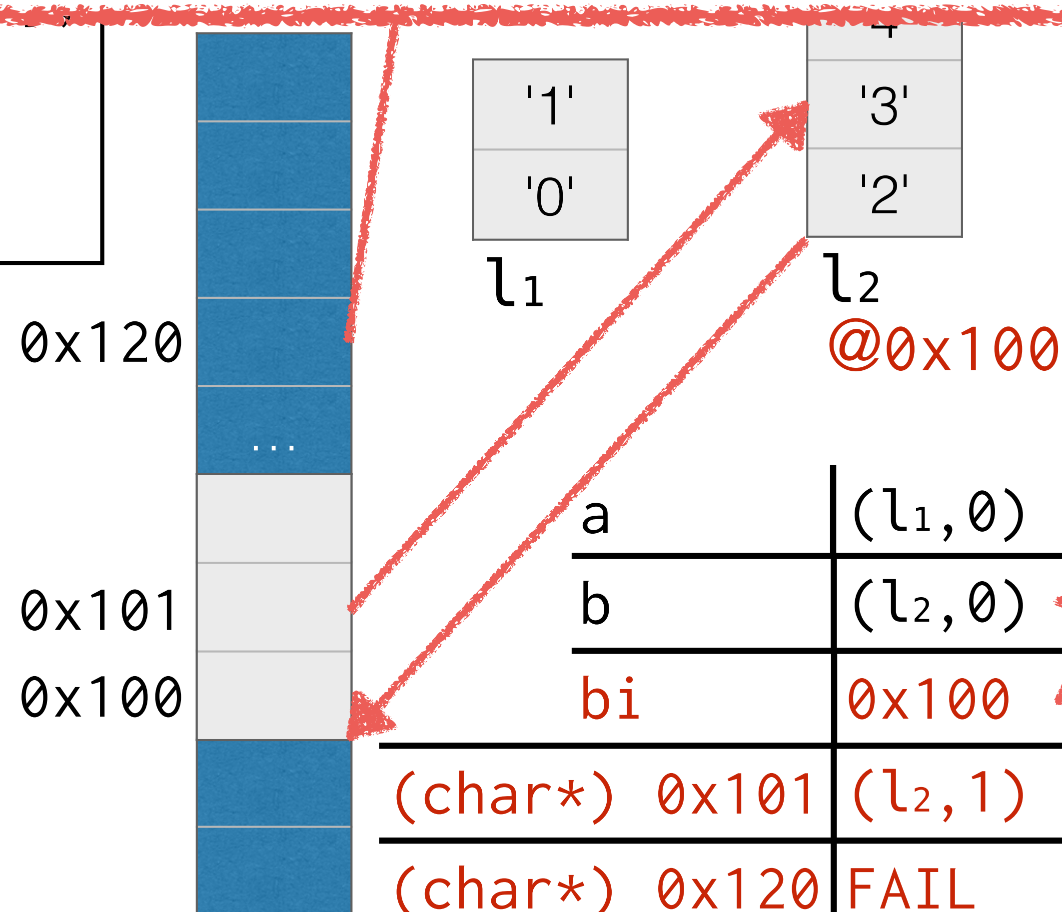
```
char a[2] = {'0','1'};
char b[3] = {'2','3','4'};
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



Our Model:

- **Realizes blocks when casting to integer**
- **Casts back to corresponding blocks**

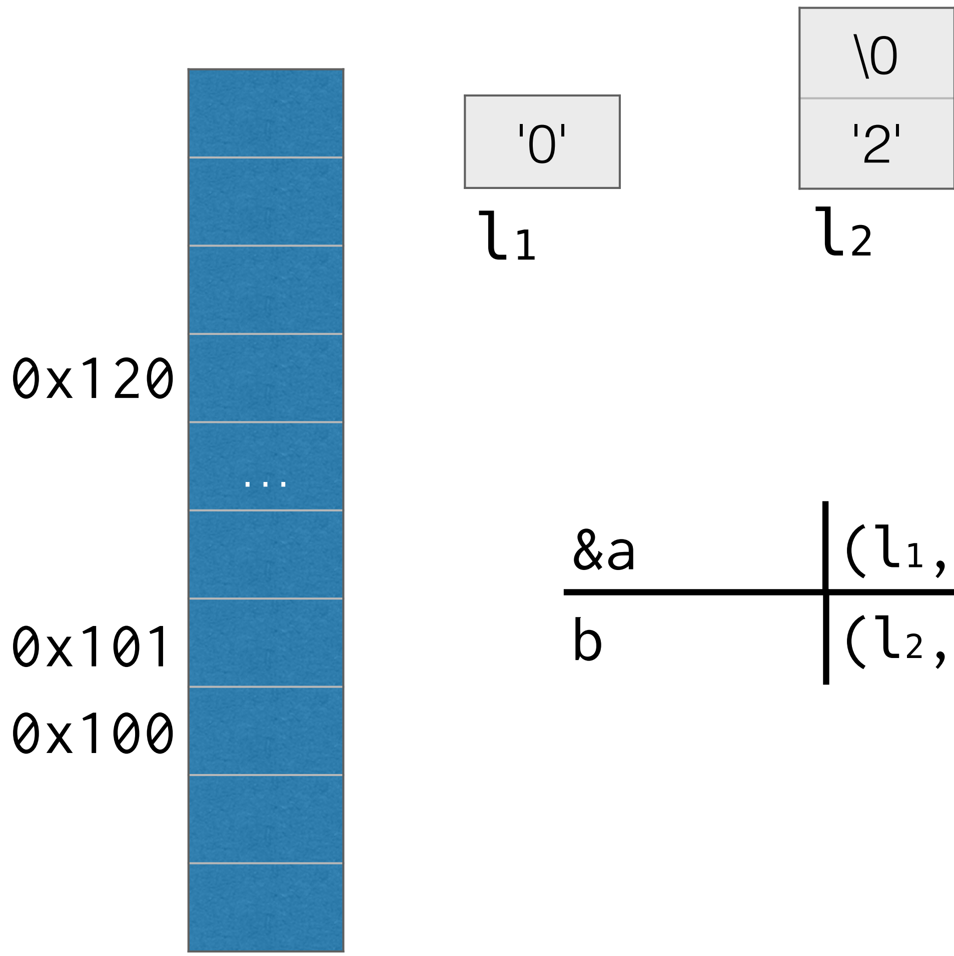
```
bi = (uintptr_t) b;
p1 = (char*) 0x101;
p2 = (char*) 0x120;
```



a	(l₁,0)
b	(l₂,0)
bi	0x100
(char*) 0x101	(l₂,1)
(char*) 0x120	FAIL

Benefits of Our Model (1/6): Still Validates Optimizations

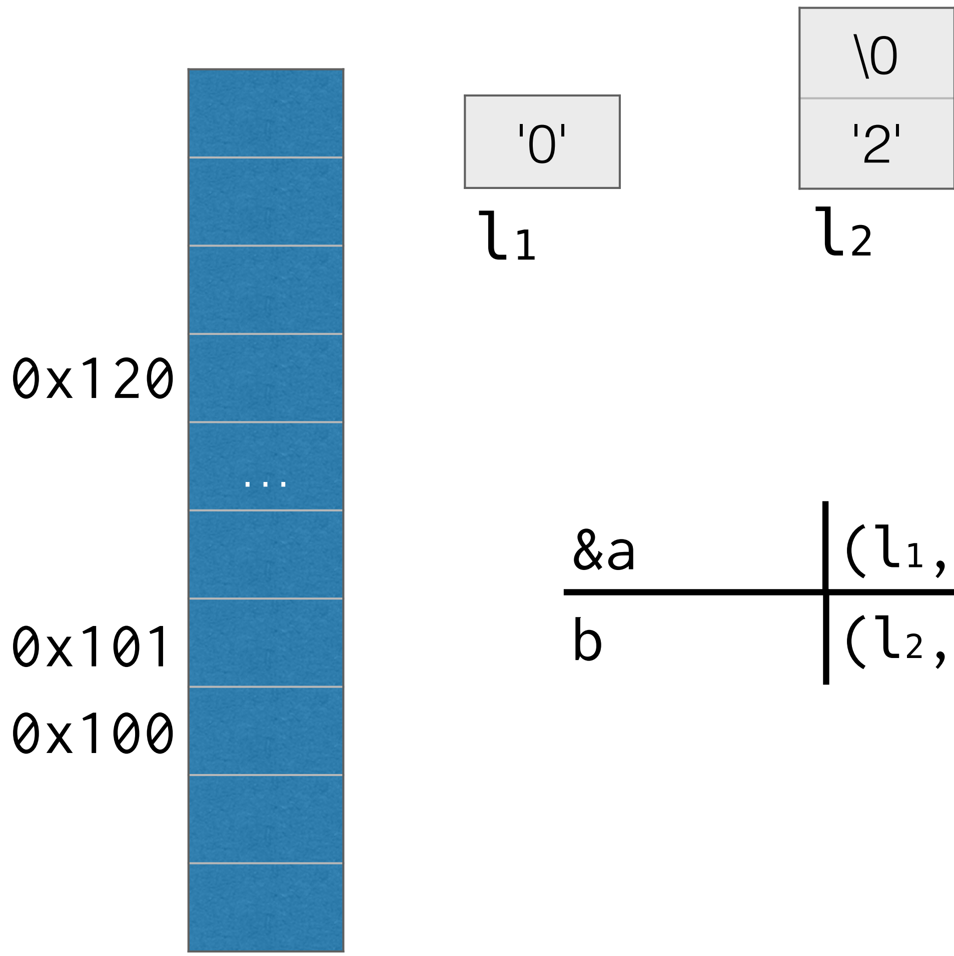
```
void g() {
 char b[2]={'2','3'};
 char* p = b + 0x20;


 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



&a	(l_1,0)
b	(l_2,0)

Benefits of Our Model (1/6):
Still Validates Optimizations

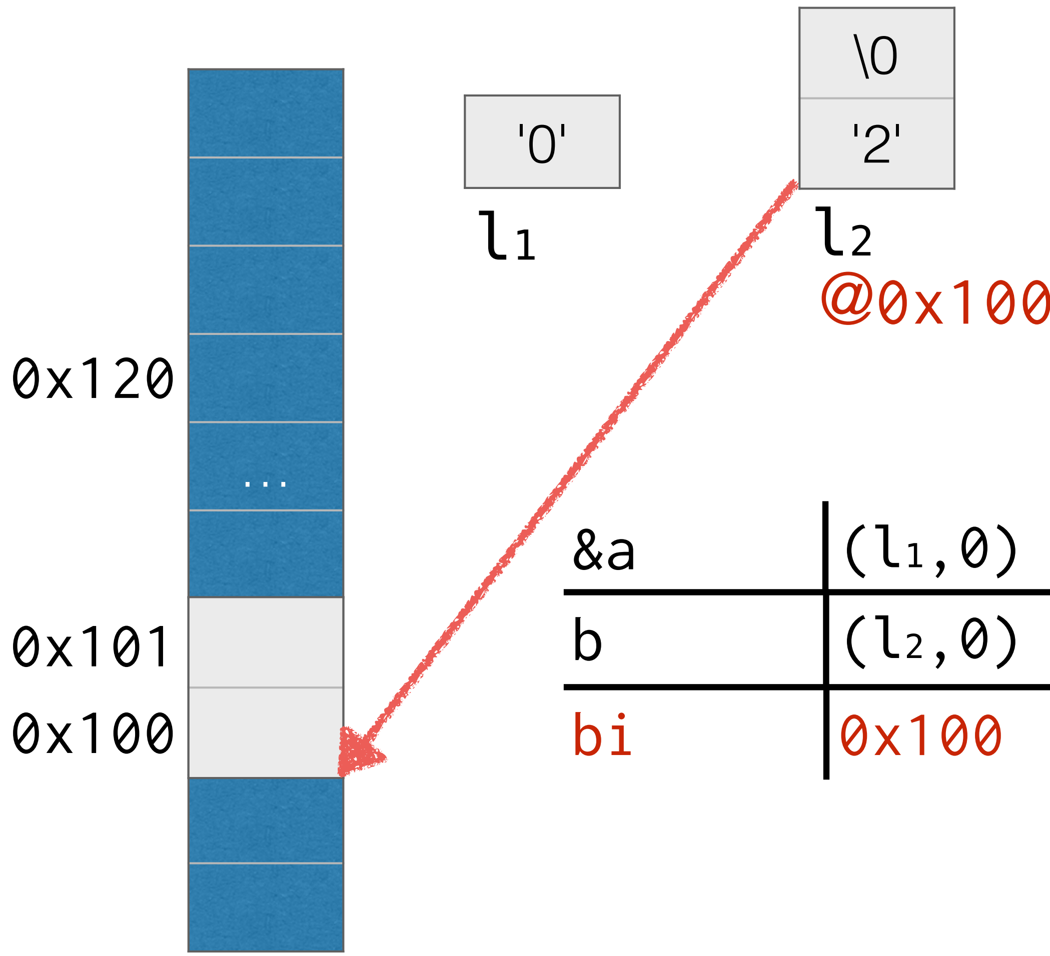
```
void g() {
 char b[2]={'2','3'};
 //char* p = b + 0x20;
 bi = (uintptr_t) b;
 p = (char*) (bi+0x20);
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



&a	$(l_1, 0)$
b	$(l_2, 0)$

Benefits of Our Model (1/6):
Still Validates Optimizations

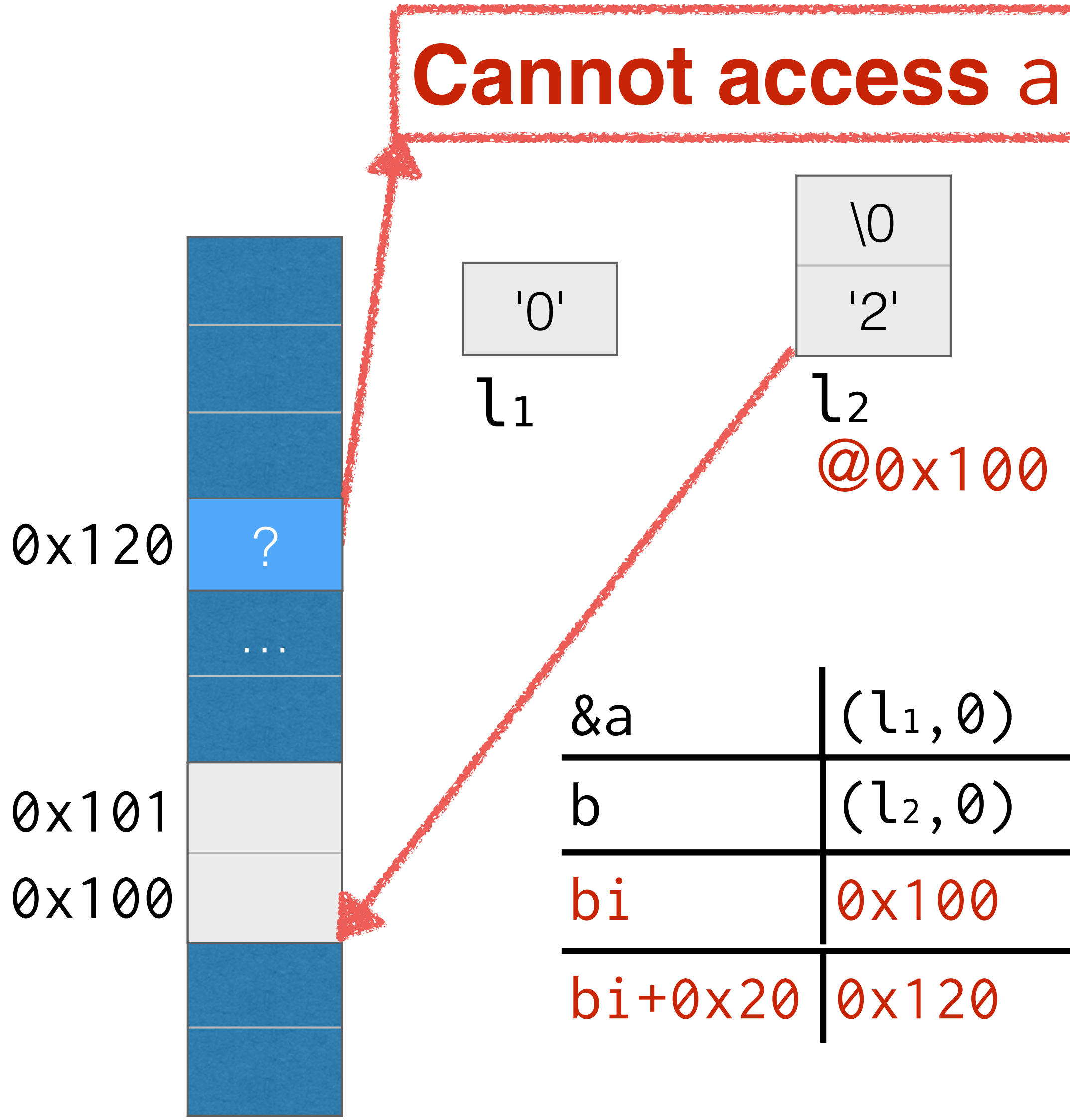
```
void g() {
 char b[2]={'2','3'};
 //char* p = b + 0x20;
 bi = (uintptr_t) b;
 p = (char*) (bi+0x20);
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



&a	(l_1,0)
b	(l_2,0)
bi	0x100

Benefits of Our Model (1/6):
Still Validates Optimizations

```
void g() {
 char b[2]={'2','3'};
 //char* p = b + 0x20;
 bi = (uintptr_t) b;
 p = (char*) (bi+0x20);
 *p = '1';
}
char f() {
 char a = '0';
 g();
 return a; // -> '0'
}
```



&a	(l1,0)
b	(l2,0)
bi	0x100
bi+0x20	0x120

Benefits of Our Model (2/6):
Fully Supports Integer-Pointer Casts

- **Pointer-to-integer casts always succeed.**
- **Integer operations on casted pointers always succeed.**

Benefits of Our Model (3/6): Simple Semantics

- Integer values are just integers w/o permission.
- Integer operations are just integer operations.

Benefits of Our Model (4/6): More Optimizations

- **Integer optimizations are allowed.**

```
int a = x - x; // -> int a = 0;
```

- **The useful code motion is allowed.**

```
int a, b;
…
if (a != b) {
  a = b;
}
```

```
int a, b;
…
if (a != b) {
}
a = b;
```

Benefits of Our Model (5/6):
Easily Applicable to Compilers

- **Just treat "casted pointers as escaped".**

DEAD CODE

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

Benefits of Our Model (5/6):
Easily Applicable to Compilers

- **Just treat "casted pointers as escaped".**

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

DEAD CODE

treated as escaped

Benefits of Our Model (5/6): Easily Applicable to Compilers

- **Just treat "casted pointers as escaped".**

```
void main() {
  int x = 0;
  uintptr_t xi = (uintptr_t) &x;
  uintptr_t i;
  for (i = 0; i < xi; ++i) {}
  if (xi != i) {
    printf("unreachable\n");
  }
  xi = i; // code motion
  int* p = (int*) xi;
  *p = 1;
  printf("%d\n", x); } // prints 1
```

DEAD CODE

treated as escaped

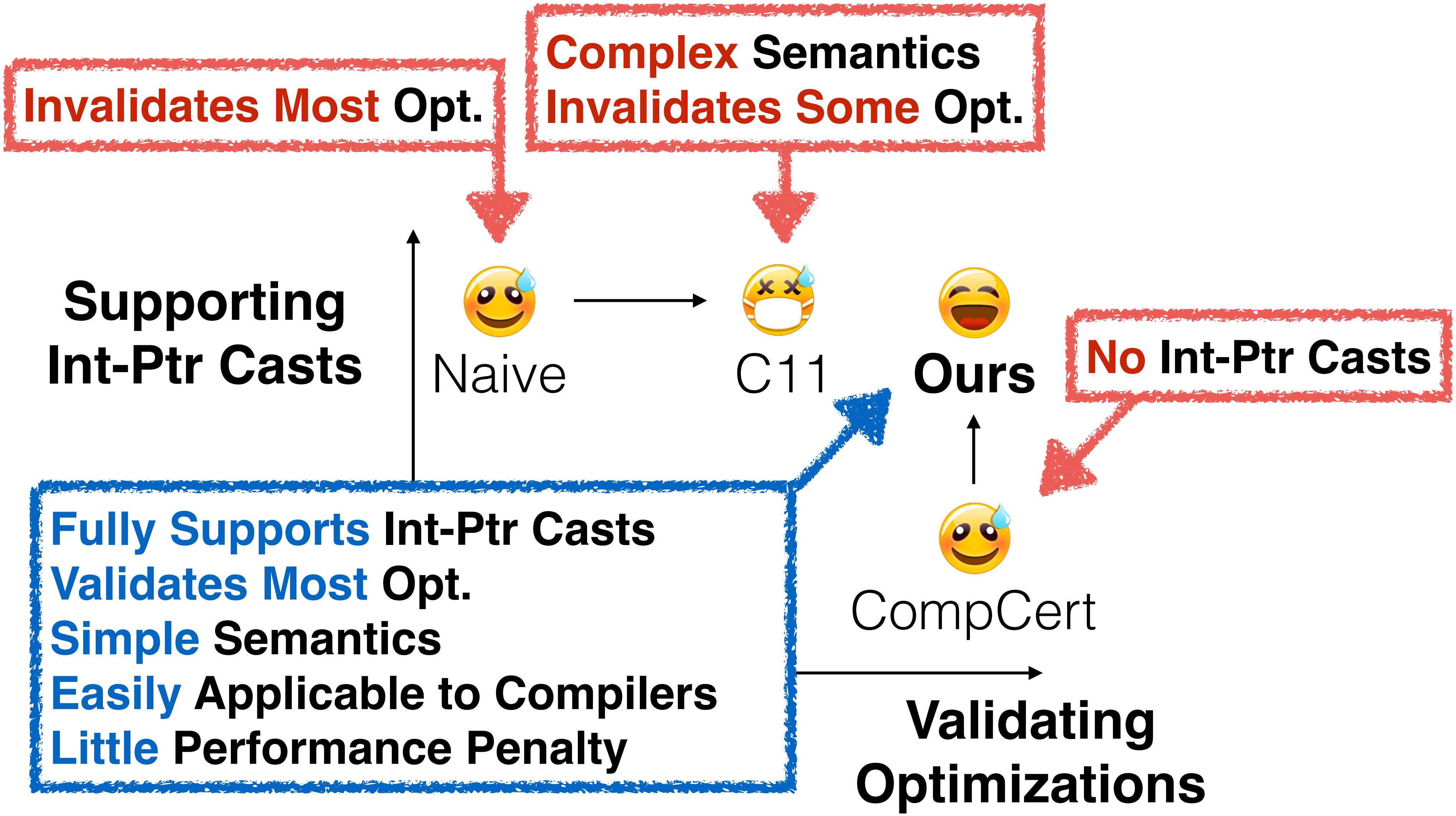
no constant propagation

Benefits of Our Model (6/6): Little Performance Penalty

- **Insignificant: performance degradation due to "casted pointers as escaped"**

  + In practice, addresses casted to integers are **global addresses**.

  + Compilers **already** treat **global addresses as escaped**.


Invalidates Most Opt.
Complex Semantics
Invalidates Some Opt.
Supporting
Int-Ptr Casts
Naive
C11
Ours
No Int-Ptr Casts
CompCert
Fully Supports Int-Ptr Casts
Validates Most Opt.
Simple Semantics
Easily Applicable to Compilers
Little Performance Penalty
Validating
Optimizations

What Else is in the Paper?

- **Formal definition of our memory model**
- **Reasoning principles for compiler verification**
- **Verification of other optimization examples**
  + Dead code elim., dead allocation elim., arithmetic optimizations, alias analysis, etc.
- **Comparison with other possible models**

Fully formalized in Coq

Application to CompCert

- **Problem**
  - Non-determinism at Ptr-to-Int Casting

- **Solution**
  - Mixed-Simulation Relation
    - Forward-Simulation at deterministic steps
    - Backward-Simulation at non-deterministic steps (only ptr-to-int casting)
    - An idea from my previous work:

Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language.
Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, Viktor Vafeiadis.
ICFP 2015

More Information

http://sf.snu.ac.kr/intptrcast

Thanks!