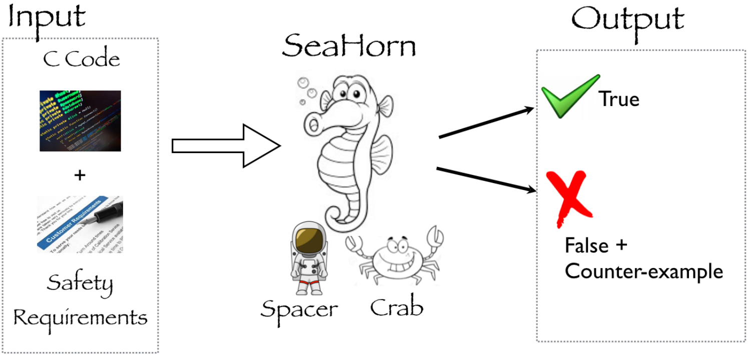
A Context-Sensitive Memory Model for Verification of C/C++ Programs

Arie Gurfinkel and Jorge A. Navas

University of Waterloo and SRI International

SAS'17, August 30th, 2017

Our Motivation



Automatic modular safety proofs on realistic C and C++ programs

Classical Memory Models for C/C++

- Byte-level model: a large array of bytes and every allocation returns a new offset in that array

$$\text{Ptr} = \text{Int} \qquad \textit{Mem} : \text{Ptr} \rightarrow \text{Byte}$$

- Untyped Block-level model: a pointer is a pair $\langle \textit{ref}, o \rangle$ where *ref* uniquely defines a memory object and o defines the byte in the object being point to

$$\text{Ptr} = \text{Ref} \times \text{Int} \qquad \textit{Mem} : \text{Ptr} \rightarrow \text{Ptr}$$

- Typed Block-level model: refines the block-level model by having a separate block for each distinct type:

$$\text{Ptr} = \text{Ref} \times \text{Int} \qquad \textit{Mem} : \text{Type} \times \text{Ptr} \rightarrow \text{Ptr}$$

Classical Memory Models for C/C++

- Byte-level model: a large array of bytes and every allocation returns a new offset in that array

$$\mathrm{Ptr} = \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Ptr} \to \mathrm{Byte}$$

- Untyped Block-level model: a pointer is a pair $\langle ref, o \rangle$ where *ref* uniquely defines a memory object and *o* defines the byte in the object being point to

$$\mathrm{Ptr} = \mathrm{Ref} \times \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Ptr} \to \mathrm{Ptr}$$

- Typed Block-level model: refines the block-level model by having a separate block for each distinct type:

$$\mathrm{Ptr} = \mathrm{Ref} \times \mathrm{Int} \qquad \mathit{Mem} : \mathrm{Type} \times \mathrm{Ptr} \to \mathrm{Ptr}$$

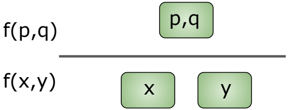
From Pointer Analysis to Verification Conditions

- Run a pointer analysis to disambiguate memory
- Produce a side-effect-free encoding by:
  - Replacing each memory object o to a logical array A_o
  - Replacing memory accesses to a pointer p (within object o) to array reads and writes over A_o
  - Each array write on A_o produces a new version of A'_o representing the array after the execution of the memory write
- Logical arrays are unbounded and the "whole array" is updated in its entirety:
  - $A[1] = 5 \;\rightarrow\; A_1 = \lambda i : i = 1 \;?\; 5 \;:\; A_0$
  - $A[k] = 7 \;\rightarrow\; A_2 = \lambda i : i = k \;?\; 7 \;:\; A_1$

VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias

f(p,q)

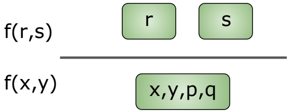
x,y,p,q

f(x,y)

VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
          int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

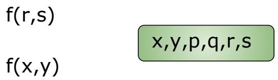
Assume p and q may alias



VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Insensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

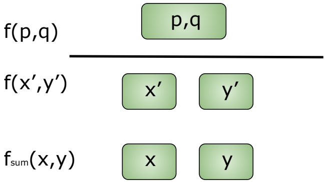
$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \mathtt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \mathtt{store}(A'_{xy}, y, 2) \\
\}
\end{array}
$$

$$
\begin{array}{l}
g(p, q, r, s, A_{pqrs}, A''_{pqrs})\{ \\
\quad f(p, q, A_{pqrs}, A'_{pqrs}) \\
\quad f(r, s, A'_{pqrs}, A''_{pqrs}) \\
\}
\end{array}
$$

VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

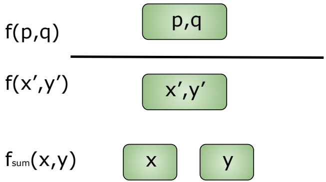
Assume p and q may alias



VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

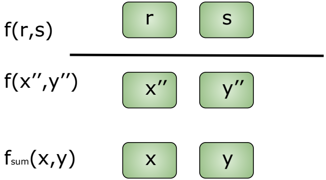
Assume p and q may alias



VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Assume p and q may alias



VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

$f(x, y, A_x, A_y, A'_x, A'_y)\{$
$\quad A'_x = \mathtt{store}(A_x, x, 1)$
$\quad A'_y = \mathtt{store}(A_y, y, 2)$
$\}$

$g(p, q, r, s, A_{pq}, A_r, A_s, A'_{pq}, A'_r, A'_s)\{$
$\quad f(p, q, A_{pq}, A_{pq}, A'_{pq}, A'_{pq})$
$\quad f(r, s, A_r, A_s, A'_r, A'_s)$
$\}$

VCs Using a Context-Sensitive Pointer Analysis

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Verification conditions:

$$
\begin{aligned}
& f(x, y, A_x, A_y, A'_x, A'_y)\{ \\
& \quad A'_x = \texttt{store}(A_x, x, 1) \\
& \quad A'_y = \texttt{store}(A_y, y, 2) \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
& g(p, q, r, s, A_{pq}, A_r, A_s, A'_{pq}, A'_r, A'_s)\{ \\
& \quad f(p, q, A_{pq}, A_{pq}, A'_{pq}, A'_{pq}) \\
& \quad f(r, s, A_r, A_s, A'_r, A'_s) \\
& \}
\end{aligned}
$$

A direct VC encoding is unsound:

First call to f: $A'_{pq} = \texttt{store}(A_{pq}, p, 1)$ and $A'_{pq} = \texttt{store}(A_{pq}, q, 2)$

The update of `p` is lost!

Ensuring Sound VCs using a CS Pointer Analysis

- Arbitrary CS pointer analysis cannot be directly leveraged for modular verification

- They must satisfy this Correctness Condition (CC):

  "No two disjoint memory objects modified in a function can be aliased at any particular call site"

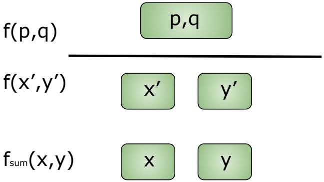
- Observed by Reynolds'78, Moy's PhD thesis'09, and many others

- Proposed solutions:
  - ignore context-sensitivity: SMACK and Cascade
  - generate contracts that ensure CC holds, otherwise reject programs: Frama-C + Jessie plugin

Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

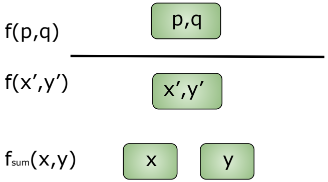
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

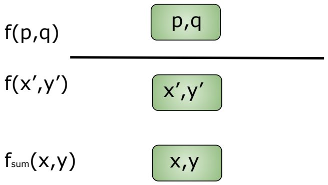
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
        int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

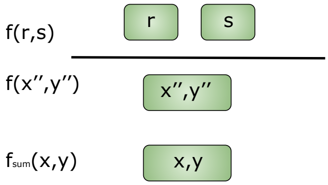
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

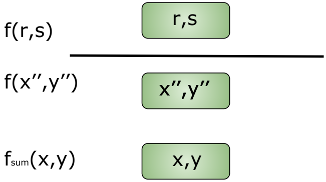
Assume p and q may alias



Ensuring Sound Modular VC Generation: Our Solution

Assume p and q may alias

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
         int* r,int* s) {
  f(p,q);
  f(r,s);
}
```



Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
       int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Sound verification conditions:

$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \texttt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \texttt{store}(A'_{xy}, y, 2) \\
\} \\
g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs})\{ \\
\quad f(p, q, A_{pq}, A'_{pq}) \\
\quad f(r, s, A_{rs}, A'_{rs}) \\
\}
\end{array}
$$

Ensuring Sound Modular VC Generation: Our Solution

```
void f(int* x,int* y) {
  *x = 1;
  *y = 2;
}

void g(int* p,int* q,
          int* r,int* s) {
  f(p,q);
  f(r,s);
}
```

Sound verification conditions:

$$
\begin{array}{l}
f(x, y, A_{xy}, A''_{xy})\{ \\
\quad A'_{xy} = \texttt{store}(A_{xy}, x, 1) \\
\quad A''_{xy} = \texttt{store}(A'_{xy}, y, 2) \\
\} \\
g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs})\{ \\
\quad f(p, q, A_{pq}, A'_{pq}) \\
\quad f(r, s, A_{rs}, A'_{rs}) \\
\}
\end{array}
$$

Good compromise:

context-sensitive: calls to f do not merge {p,q} and {r,s}

ensure that CC holds!

Field- and Array-Sensitive Pointer Analysis

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}

void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```

Our pointer analysis infers:

1. `&a[0]` points to an object O_A which has ≥ 1 elements of size of a pointer
2. O_A points to another object O_L with 0 and 4 offsets

Similar pointer analyses do not distinguish O_A from O_L

Our contributions

We present a new pointer analysis for verification of C/C++ that:

1. is context-, field-, and array-sensitive
2. has been implemented and publicly available

   `https://github.com/seahorn/sea-dsa`

3. has been evaluated on flight control components written in C++ and SV-COMP benchmarks in C

Concrete Semantics

- A concrete cell is a pair of an object reference and offset
- A concrete points-to graph $g \in \mathcal{G}_{\mathbb{C}}$ is a triple $\langle V, E, \sigma \rangle$:

$$V \subseteq \mathcal{C}_{\mathbb{C}} \quad E \subseteq \mathcal{C}_{\mathbb{C}} \times \mathcal{C}_{\mathbb{C}} \quad \sigma : \mathcal{V}_{\mathcal{P}} \mapsto \mathcal{C}_{\mathbb{C}}$$

- A concrete state is a triple $\langle g, \pi, pc \rangle$ where

$$g \in \mathcal{G}_{\mathbb{C}} \quad \pi : \mathcal{V}_{\mathcal{I}} \mapsto \mathbb{Z} \quad pc \in \mathbb{L}$$

- **malloc** returns a fresh memory object

Concrete Semantics: Assumptions

1. Freed memory is not reused:

```
int *p = (int*) malloc(..);
int *q = p;
free(p);
int *r = (int*) malloc(..)
```

   it assumes that `r` cannot alias with `q`

2. It does not distinguish between valid and invalid pointers:

```
int *p = (int*) malloc(..);
free(p);
int *q = (int*) malloc(..);
if (p == q) *p=0;
```

   it assumes no null dereference

Abstract Semantics

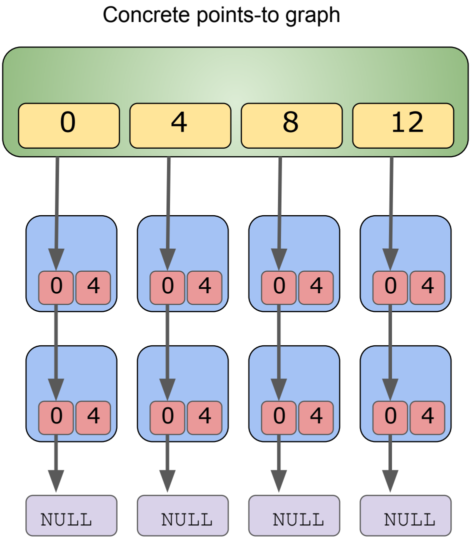
- An abstract cell is a pair of an abstract object and byte offset
- An abstract object has an identifier and:
  1. is_sequence: unknown sequence of consecutive bytes
  2. is_collapsed: all outgoing cells have been merged
  3. size in bytes (see paper for details)
- An abstract points-to graph $\mathcal{G}_{\mathbb{A}}$ is a triple $\langle V, E, \sigma \rangle$:

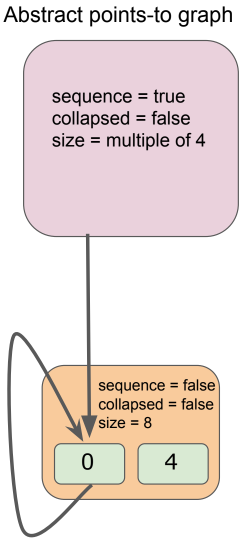
$$V \subseteq \mathcal{C}_{\mathbb{A}} \quad E \subseteq \mathcal{C}_{\mathbb{A}} \times \mathcal{C}_{\mathbb{A}} \quad \sigma : \mathcal{V}_{\mathcal{P}} \mapsto \mathcal{C}_{\mathbb{A}}$$

  The number of abstract objects is finite

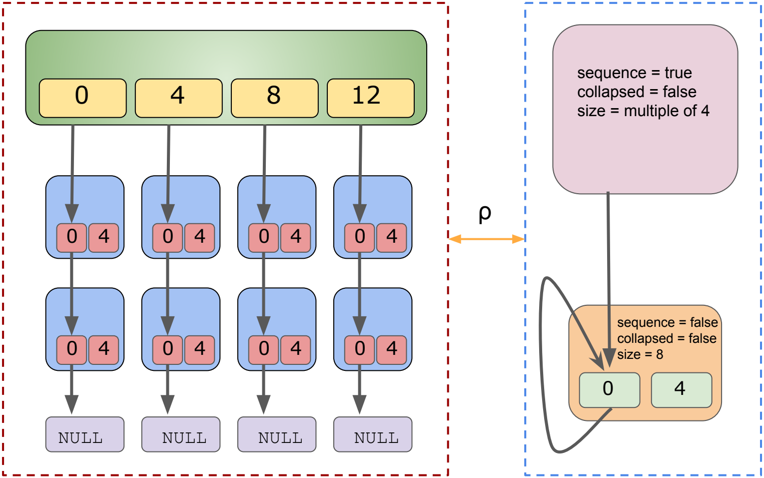
- An abstract state is represented by an abstract points-to graph
  - it does not keep track of an environment for integer variables
  - it is flow-insensitive

Concrete vs Abstract points-to Graphs

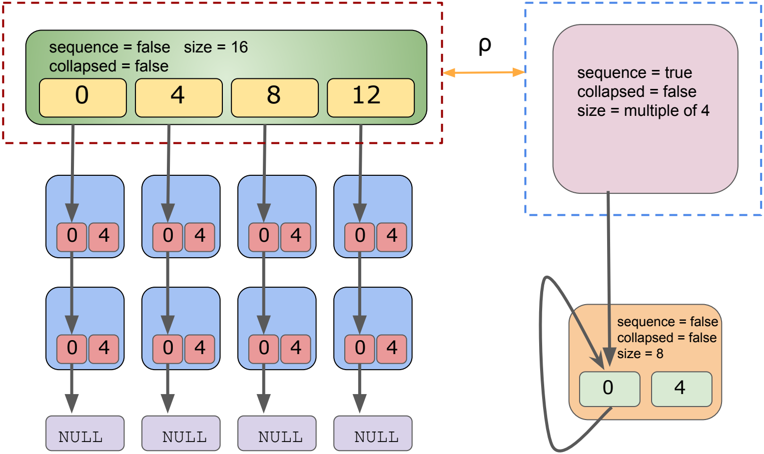




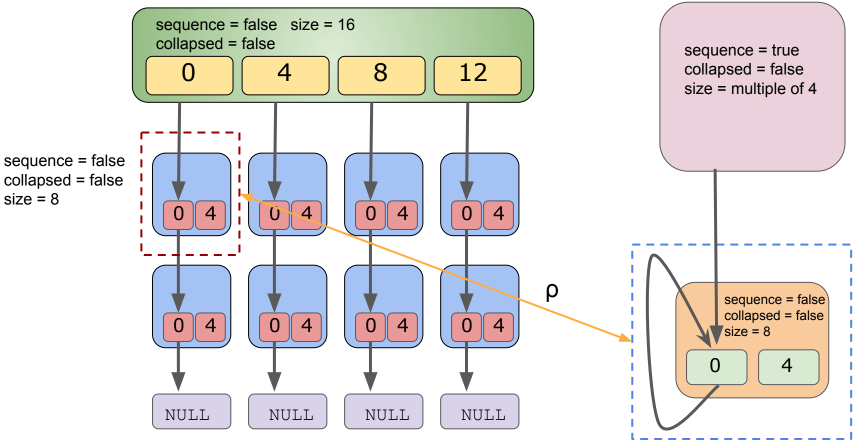
Simulation Relation between Graphs



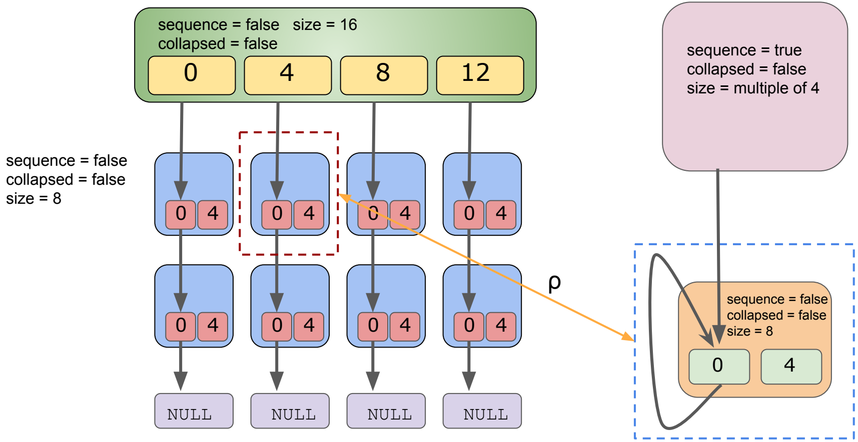
Simulation Relation between Graphs



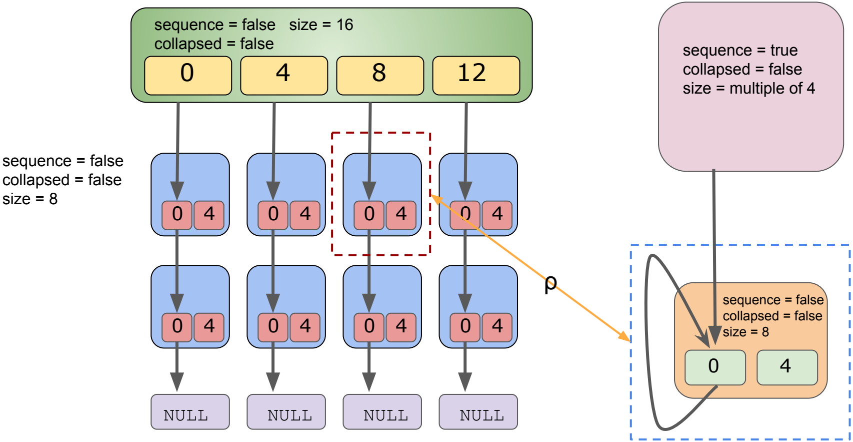
Simulation Relation between Graphs



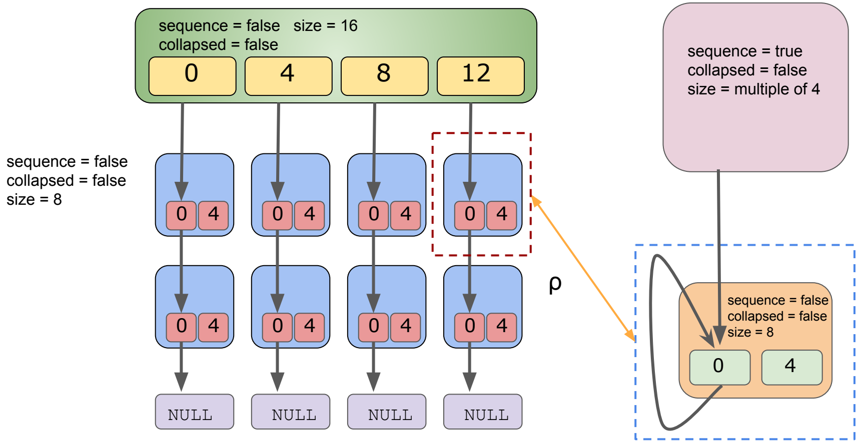
Simulation Relation between Graphs



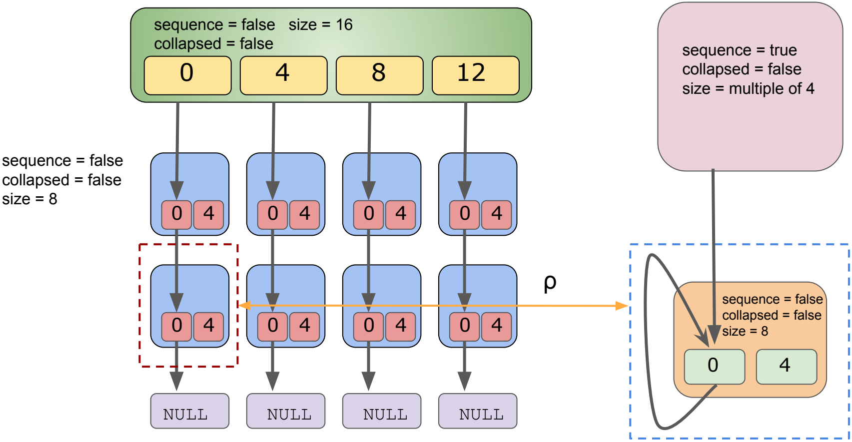
Simulation Relation between Graphs



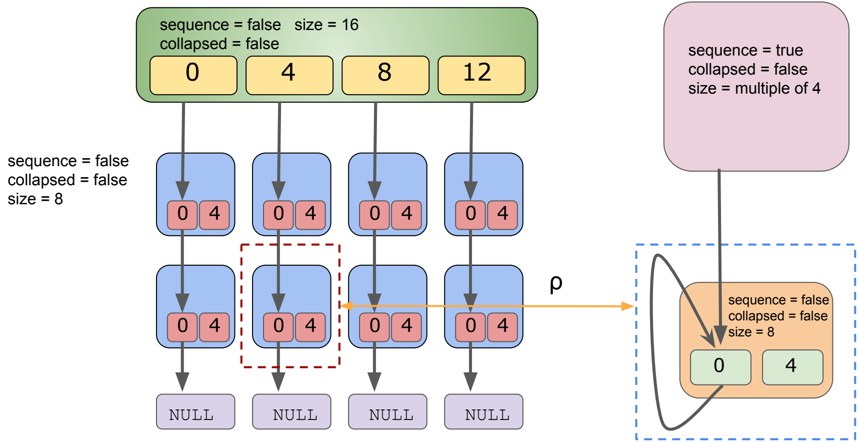
Simulation Relation between Graphs



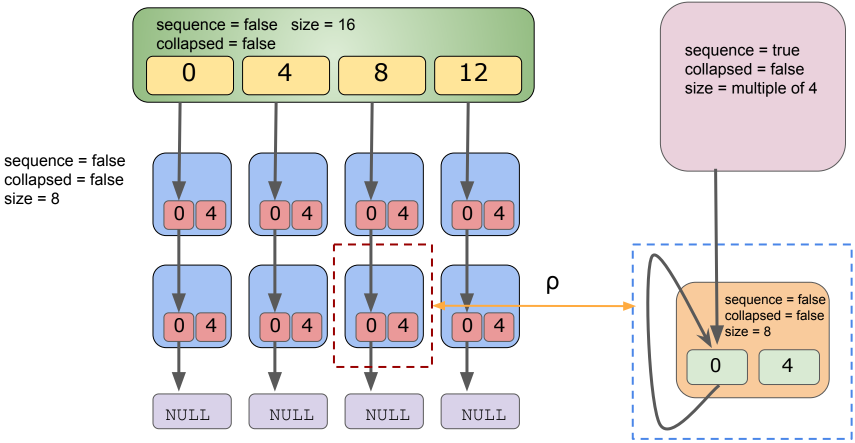
Simulation Relation between Graphs



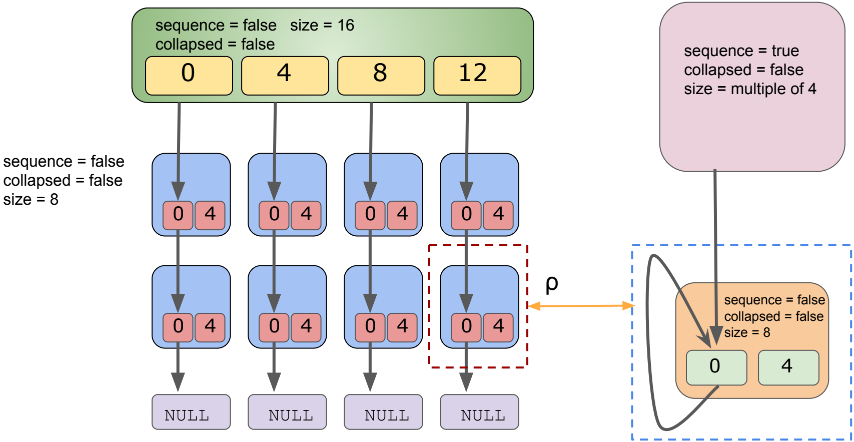
Simulation Relation between Graphs



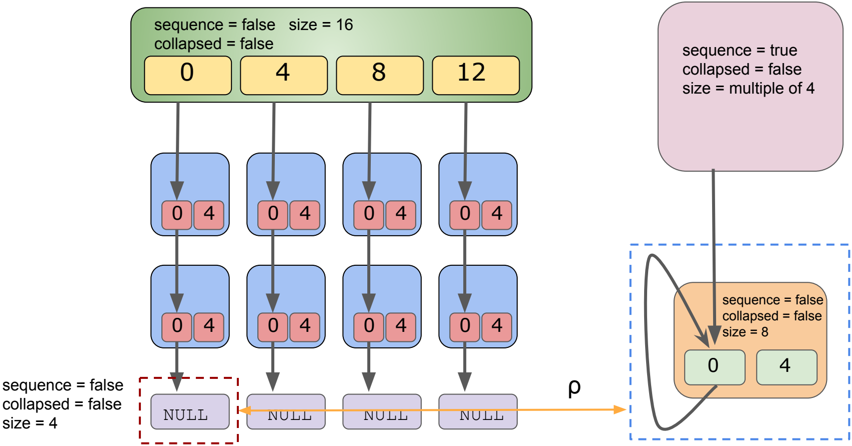
Simulation Relation between Graphs



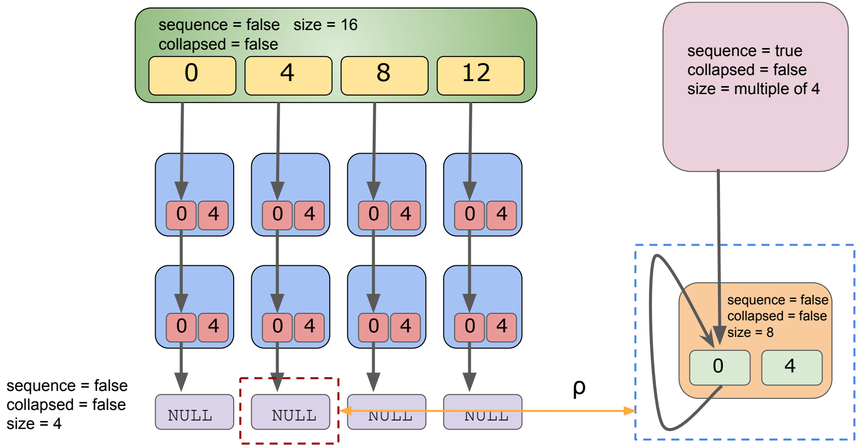
Simulation Relation between Graphs



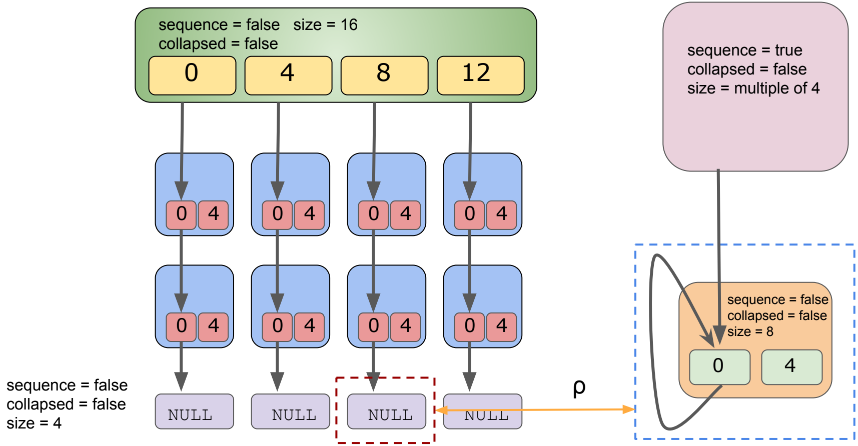
Simulation Relation between Graphs



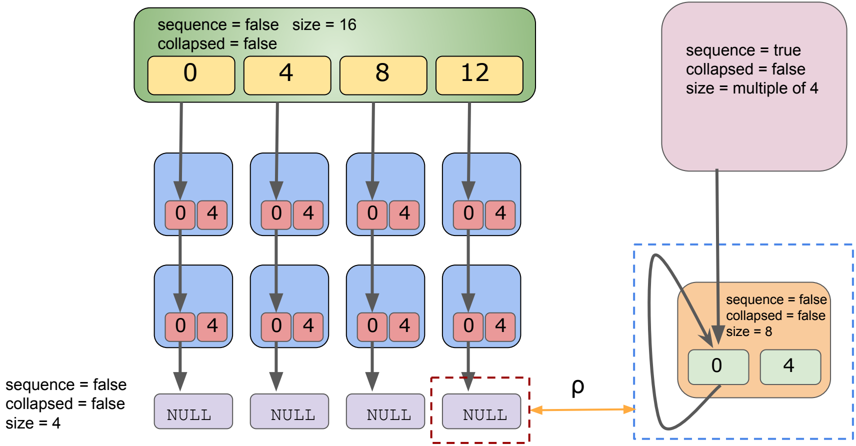
Simulation Relation between Graphs



Simulation Relation between Graphs



Simulation Relation between Graphs



Simulation Relation between Graphs

- $\gamma : \mathcal{G}_{\mathbb{A}} \mapsto 2^{\mathcal{G}_{\mathbb{C}}}$ defined as

$$\gamma(g_a) = \{g_c \in \mathcal{G}_{\mathbb{C}} \mid g_c \text{ simulated by } g_a\}$$

- It defines also an ordering between abstract graphs $g, g' \in \mathcal{G}_{\mathbb{A}}$

$$g \sqsubseteq_{\mathcal{G}_{\mathbb{A}}} g' \text{ if and only if } g \text{ is simulated by } g'$$

- It will play an essential role during the context-sensitive analysis (later in this talk)

Intra-Procedural Pointer Analysis

- Based on field-sensitive Steensgaard's
- Key operation: cell unification
- Ensure $c_1 = (n_1, o_1)$ and $c_2 = (n_2, o_2)$ are the same address
- If $o_1 < o_2$ then (other case symmetric)

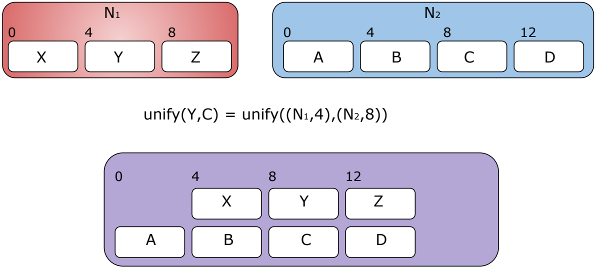
  map $(n_1, 0)$ to $(n_2, o_2 - o_1)$

  $(n_1, o_1) = (n_2, o_2 - o_1 + o_1) = (n_2, o_2)$

  unify each (n_1, o_k) with $(n_2, o_2 - o_1 + o_k)$

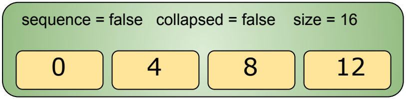
Intra-Procedural Pointer Analysis

- Based on field-sensitive Steensgaard's
- Key operation: cell unification
- Ensure $c_1 = (n_1, o_1)$ and $c_2 = (n_2, o_2)$ are the same address
- If $o_1 < o_2$ then (other case symmetric)
  map $(n_1, 0)$ to $(n_2, o_2 - o_1)$
  $(n_1, o_1) = (n_2, o_2 - o_1 + o_1) = (n_2, o_2)$
  unify each (n_1, o_k) with $(n_2, o_2 - o_1 + o_k)$



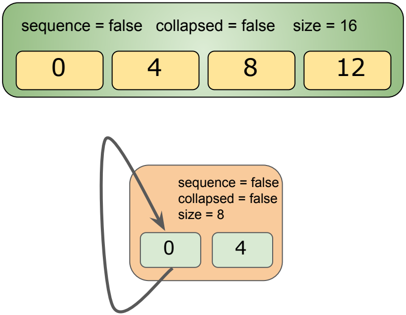
Array-Sensitivity

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



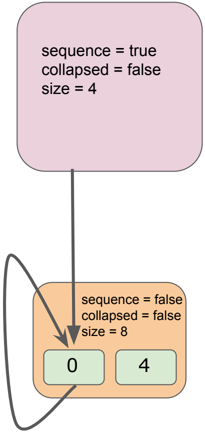
Array-Sensitivity

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



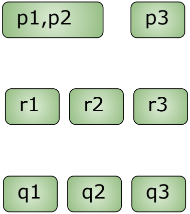
Array-Sensitivity

```
typedef struct list{
  struct list *n;
  int e;
} ll;

ll* mkList(int s,int e){
 if (s <= 0)
   return NULL;
 ll*p=malloc(sizeof(ll));
 p->e=e;
 p->n=mkList(s-1,e);
 return p;
}
#define N 4
void main(){
 ll* a[N];
 int i;
 for(i=0;i<N;++i)
   a[i] = mkList(M,0);
}
```



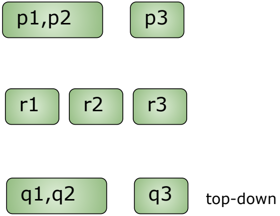
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



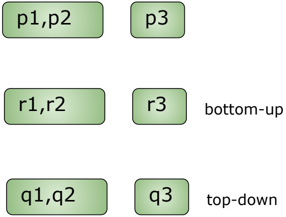
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



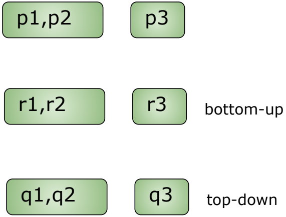
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



Context-Sensitive Pointer Analysis

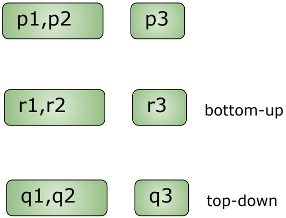
```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



- Next, `h`'s callsites and callsites where `h` is called must be re-analyzed, and so on

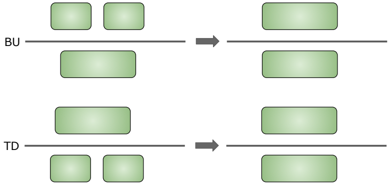
Context-Sensitive Pointer Analysis

```
void g(...) {
  f(p1,p2,p3);
}
void h(...) {
  f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
  ...
}
```



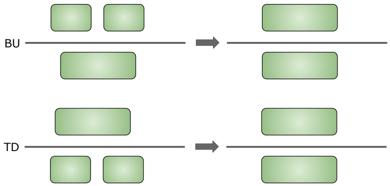
- Next, h's callsites and callsites where h is called must be re-analyzed, and so on
- In general, after an unification we need to re-analyze:
  - if top-down: callsites with same callee and callsites within the callee
  - if bottom-up: callsites with same caller and callsites within the caller
- However, no need to re-analyze the whole function!
- Fixpoint over all callsites until no more bottom-up or top-down unifications

Bottom-Up and Top-Down Unifications



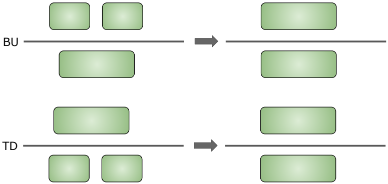
Q: How to decide whether BU, TD or no more unifications?

Bottom-Up and Top-Down Unifications



Q: How to decide whether BU, TD or no more unifications?
A: Simulation relation!

Bottom-Up and Top-Down Unifications



Q: How to decide whether BU, TD or no more unifications?
A: Simulation relation!

Build a simulation relation ρ between callee and caller graphs:

1. if ρ is not a function then BU
2. else if ρ is a function but not injective then TD
3. else ρ is an injective function then do nothing

Context-Sensitive Pointer Analysis: All Pieces Together

1. for each function in reverse topological order of the call graph

   compute summary

2. for each callsite

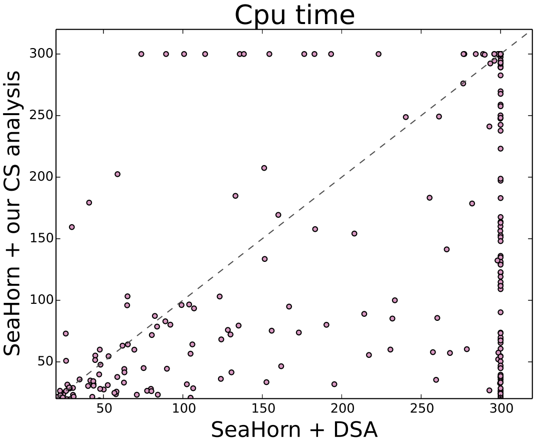
   clone callee's summary into the caller graph and unify formal/actual cells

3. apply BU and TD unifications until CC holds for all callsites

Experiments

- Integrated the pointer analysis in SeaHorn
- The pointer analysis is used during VC generation
- Compared SeaHorn verification time using:
  - (CI) DSA Pointer analysis from LLVM PoolAlloc project
  - Our pointer analysis

Experiments on SV-COMP C Programs



- 2000 benchmarks from SV-COMP DeviceDrivers64 category
- Verification time with timeout of 5m and 4GB memory limit
- With our analysis SeaHorn proved 81 more programs

(Ongoing) C++ Case Study

Goal:

Verify absence of buffer overflows on the flight control system of the Core Autonomous Safety Software (CASS) of an Autonomous Flight Safety System

- 13,640 LOC (excluding blanks/comments) written in C++ using standard C++ 2011 and following MISRA C++ 2008
- It follows an object-oriented style and makes heavy use of dynamic arrays and singly-linked lists

	#Objects	#Collapsed	Max. Density	% Proven
Sea + DSA	258	49%	80%	13
Sea + our CS	12,789	4%	13%	21

Conclusions

- Modular proofs require context-sensitive heap reasoning
- We adopted a very high-level memory model that can still express low-level C/C++ features such as:
  - pointer arithmetic, pointer casts and type unions
- We presented a scalable field-,array-,context-sensitive pointer analysis tailored for VC generation
  - A simulation relation between points-to graphs plays a major role in the analysis of function calls
- It can produce a finer-grained partition of memory that often results in faster verification times