A Context Sensitive Memory Model for Software Model Checking

Arie Gurfinkel

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

http://ece.uwaterloo.ca/~agurfink

joint work with Jorge A. Navas (SRI)
Automated (Software) Verification

Program and/or model

Alan M. Turing. 1936: “Undecidable”

Alan M. Turing. ”Checking a large routine” 1949

How can one check a routine in the sense of making sure that it is right?

The programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.
Automated Software Analysis

Model Checking

[Clarke and Emerson, 1981]

Abstract Interpretation

[Queille and Sifakis, 1982]

Symbolic Execution

[King, 1976]

[Cousot and Cousot, 1977]
SeaHorn

A fully automated verification framework for LLVM-based languages.

http://seahorn.github.io
Algorithmic Logic-based Program Verification

Low-Level Bounded Model Checking (BMC)
• decide whether a low level program/circuit has an execution of a given length that violates a safety property
• effective decision procedure via encoding to propositional SAT

High-Level (Word-Level) Bounded Model Checking
• decide whether a program has an execution of a given length that violates a safety property
• efficient decision procedure via encoding to SMT

What is an SMT-like equivalent for Safety Verification?
• Logic: SMT-Constrained Horn Clauses
• Decision Procedure: Spacer / GPDR
  – extend IC3/PDR algorithms from Hardware Model Checking
Algorithmic Logic-Based Verification

Program + Spec

Verification Condition (in Logic)

Decision Procedure

Yes

No

Safety Properties

Constrained Horn Clauses

Spacer
Horn Clauses for Program Verification

De Angelis et al. Verifying Array Programs by Transforming Verification Conditions. VMCAI'14

Bjørner, Gurfinkel, McMillan, and Rybalchenko: Horn Clause Solvers for Program Verification
Horn Clauses for Concurrent / Distributed / Parameterized Systems

For assertions $R_1, \ldots, R_N$ over $V$ and $E_1, \ldots, E_N$ over $V, V'$,

CM1: \[ \text{init}(V) \quad \rightarrow \quad R_1(V) \]

CM2: \[ R_i(V) \land \rho_i(V, V') \quad \rightarrow \quad R_i(V') \]

CM3: \[ \left( \bigvee_{i \in 1..N} \bigwedge_{j \neq i} R_i(V) \land \rho_i(V, V') \right) \quad \rightarrow \quad E_j(V, V') \]

CM4: \[ R_i(V) \land E_j(V, V') \land \rho_j^c(V, V') \quad \rightarrow \quad R_i(V') \]

CM5: \[ R_1(V) \land \cdots \land R_N(V) \land \text{error}(V) \quad \rightarrow \quad \text{false} \]

multi-threaded program $P$ is safe

Rybalchenko et al. Synthesizing Software Verifiers from Proof Rules. PLDI'12

Hoenicke et al. Thread Modularity at Many Levels. POPL'17

Hojjat et al. Horn Clauses for Communicating Timed Systems. HCVS'14

Gurfinkel et al. SMT-Based Verification of Parameterized Systems. FSE 2016

Figure 4: Horn constraints encoding a homogeneous infinite system with the help of a $k$-indexed invariant. $S_k$ is the symmetric group on $\{1, \ldots, k\}$, i.e., the group of all permutations of $k$ numbers; as an optimisation, any generating subset of $S_k$, for instance transpositions, can be used instead of $S_k$. In (10), we define $r = \max\{m, k\}$.

Figure 3: $VC_2(T)$ for two-quantifier invariants.
Logic-based Algorithmic Verification

- Simulink
- Lustre
- T2
- Java
- C/C++
- concurrent/distributed systems
- CPR
- Termination for C

Spacer
Architecture of Seahorn

**Clang**
- C/C++
- LLVM Opt:
  - SSA
  - DCE
  - Peephole
  - CFG Simplification
- Devirtualization and Exception Lowering
- Property Instr:
  - Buffer overflow
  - Null dereferences
- Slicing Assertions

**LLVM bitcode**
- LLVM Opt: SSA, DCE, Peephole, CFG Simplification

**Horn Clauses**
- PDR/IC3-based Model checking
- Abstract Interp.:
  - Intervals
  - DBMs
  - LDDs
- Template-based (Houdini)
- BMC bitvectors

**Front-end**
- Heap Abstraction
  - Precision:
    - Integers
    - Floating point
    - Pointers
    - Memory contents
- Array Abstraction

**Middle-end**
- Property Instr:
  - Buffer overflow
  - Null dereferences

**Back-end**
- BMC bitvectors
SeaHorn Workflow

Property Spec

Verification Environment

Property Checker

Code Under Analysis (CUA)

Verification Problem (VP)

SeaHorn

Good + Verification Certificate (Cert)

Bad + Counterexample (CEX)

TestGen

Test harness (Test)
SeaHorn workflow components

Code Under Analysis (CUA)
- code being analyzed. Device driver, component, library, etc.

Verification environment
- stubs for the environment with which CUA interacts
- e.g., libc, memcpy, malloc, OS system calls, user input, socket, file, ...

Property Checker
- static instrumentation of a program with a monitor that indicates when an error has happened
- similar to dynamic sanitizers, but can use verifier-specific API to perform symbolic actions
- property spec is specific to a property checker

Verification Problem
- a prepared instance of program with embedded assertions, potentially simplified by abstracting away irrelevant parts of execution

Test Gen
- generates a test harness that includes all stubs and stimuli to guide CUA to a property failure discovered by the verifier
Developing a Static Property Checker

A static property checker is similar to a dynamic checker
• e.g., clang sanitizer (address, thread, memory, etc.)

A significant development effort for each new property
• new specialized static analyses to rule out trivial cases
• different instrumentations have affect on performance

Developed by a domain expert
• understanding of verification techniques is useful (but not required)
• 3-6 month effort for a new property
  – but many things can be reused between similar properties
  – e.g., memory safety, null-dereference, taint checking, use-after-free, etc.

SeaHorn property checkers:
• memory safety (out of bound uses, null pointer)
  – ongoing work to improve scalability and usability
• taint analysis (being developed by Princeton)
Classical Memory Models for C/C++

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

\[ \text{Ptr} = \text{Int} \quad \text{Mem} : \text{Ptr} \rightarrow \text{Byte} \]

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

\[ \text{Ptr} = \text{Ref} \times \text{Int} \quad \text{Mem} : \text{Ptr} \rightarrow \text{Ptr} \]

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

\[ \text{Ptr} = \text{Ref} \times \text{Int} \quad \text{Mem} : \text{Type} \times \text{Ptr} \rightarrow \text{Ptr} \]
Classical Memory Models for C/C++

- **Byte-level** model: a large array of bytes and every allocation returns a new offset in that array
  \[ \text{Ptr} = \text{Int} \quad \text{Mem} : \text{Ptr} \rightarrow \text{Byte} \]

- **Untyped Block-level** model: a pointer is a pair \( \langle \text{ref}, o \rangle \) where \text{ref} uniquely defines a memory object and \text{o} defines the byte in the object being point to
  \[ \text{Ptr} = \text{Ref} \times \text{Int} \quad \text{Mem} : \text{Ptr} \rightarrow \text{Ptr} \]

- **Typed Block-level** model: refines the block-level model by having a separate block for each distinct type:
  \[ \text{Ptr} = \text{Ref} \times \text{Int} \quad \text{Mem} : \text{Type} \times \text{Ptr} \rightarrow \text{Ptr} \]
Let $\mathcal{P}$ be a property of an array segment $\{A+1, \ldots, A+h\}$

Let $q$ be a pointer that is disjoint from $\{A+1, \ldots, A+h\}$

Show that $\mathcal{P}$ is true of $\{A+1, \ldots, A+h\}$ after $*q = 5$

Using byte-level model:

$$\mathcal{P}(M_0, 1, h) \land \text{disjoint}(q, 1, h) \land M_1 = \text{store}(M_0, q, 5) \Rightarrow \mathcal{P}(M_1, 1, h)$$

where

- $\text{disjoint}(q, l, h) = q + \text{size}(q) \leq l \lor q \geq h$
- auxiliary lemma:

$$\forall M_i, M_j \in \text{Mem}, \forall a, b, x \in \text{Int.}(a \leq x \leq b \land M_i[x] = M_j[x]) \Rightarrow (\mathcal{P}(M_i, a, b) \Rightarrow \mathcal{P}(M_j, a, b))$$

Using block-level model:

$$\mathcal{P}(B^A, 1, h) \land B_1^q = \text{store}(B_0^q, q, 5) \Rightarrow \mathcal{P}(B^A, 1, h)$$
Our Memory Model

- Untyped block-level based memory model
- Memory objects are infinitely apart from each other
- Implicit separation given by distinctness of block references
- Cover a relevant subset of C/C++ programs that supports:
  - dynamic memory allocation
  - type unions, pointer arithmetic, pointer casts
  - inheritance, function/method calls, etc
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:
- $A[1] = 5 \rightarrow A_1 = \lambda i : i = 1 \ ? \ 5 \ : \ A_0$
- $A[k] = 7 \rightarrow A_2 = \lambda i : i = k \ ? \ 7 \ : \ A_1$
class X {
    X() { .... }
};

class Y: public X {
    Y(): X() { .... }
};

class Z: public X {
    Z(): X() { .... }
};

Y* y = new Y();
Z* z = new Z();

% Constructor for Y
_Y.c(this) {
    _X.c(this);
    ...
}

% Constructor for Z
_Z.c(this) {
    _X.c(this);
    ...
}

% Y y = new Y();
y = _Znwm(sizeof(Y));
_Y.c(y);

% Z z = new Z();
z = _Znwm(sizeof(Z));
_Z.c(z);
VCs Using a Context-Insensitive Pointer Analysis

```c
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

\[
\begin{array}{c}
f(p, q) \\
\hline
f(x, y)
\end{array}
\]

`p, q`
VCs Using a Context-Insensitive Pointer Analysis

Assume $p$ and $q$ may alias

```c
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);
}
```

```c
f(p, q)  \quad x, y, p, q
f(x, y)
```
VCs Using a Context-Insensitive Pointer Analysis

```c
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(r, s) \hspace{1cm} r \hspace{1cm} s

\[ f(x, y) \hspace{1cm} x, y, p, q \]
VCs Using a Context-Insensitive Pointer Analysis

```c
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(r, s)`
- `x, y, p, q, r, s`
- `f(x, y)`
VCs Using a Context-Insensitive Pointer Analysis

```c
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:

```c
f(x, y, A_{xy}, A'_{xy}){
    A'_{xy} = store(A_{xy}, x, 1)
    A''_{xy} = store(A'_{xy}, y, 2)
}

g(p, q, r, s, A_{pqrs}, A'_{pqrs}){
    f(p, q, A_{pqrs}, A'_{pqrs})
    f(r, s, A'_{pqrs}, A''_{pqrs})
}
```
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

\[
\begin{align*}
    f(p, q) & \quad p, q \\
    f(x', y') & \quad x', y' \\
    f_{\text{sum}}(x, y) & \quad x \quad y
\end{align*}
\]
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

\[
f(p, q) \quad \rightarrow \quad p, q
\]

\[
f(x', y') \quad \rightarrow \quad x', y'
\]

\[
f_{sum}(x, y) \quad \rightarrow \quad x, y
\]
VCs Using a Context-Sensitive Pointer Analysis

```c
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

- \( f(r, s) \)
- \( f(x'', y'') \)
- \( f_{sum}(x, y) \)
VCs Using a Context-Sensitive Pointer Analysis

```c
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:

\[ f(x, y, A_x, A_y, A'_x, A'_y) \]  
\[ A'_x = \text{store}(A_x, x, 1) \]  
\[ A'_y = \text{store}(A_y, y, 2) \]

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

\[
f(x, y, A_x, A_y, A'_x, A'_y)\
    A'_x = \text{store}(A_x, x, 1)\
    A'_y = \text{store}(A_y, y, 2)\
\]

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

A direct VC encoding is **unsound**:  
First call to \( f \):  \( A'_{pq} = \text{store}(A_{pq}, p, 1) \)  and  \( A'_{pq} = \text{store}(A_{pq}, q, 2) \)  
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
Assume $p$ and $q$ may alias

```c
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

\[
\begin{align*}
\text{void } f(\text{int* } x, \text{int* } y) & \{ \\
& *x = 1; \\
& *y = 2; \\
\} \\
\text{void } g(\text{int* } p, \text{int* } q, \text{int* } r, \text{int* } s) & \{ \\
& f(p, q); \\
& f(r, s); \\
\}
\end{align*}
\]
Assume \( p \) and \( q \) may alias

\[
\begin{align*}
\text{void } f(\text{int* } x, \text{int* } y) \{ \\
  *x &= 1; \\
  *y &= 2; \\
\} \\
\text{void } g(\text{int* } p, \text{int* } q, \\
  \text{int* } r, \text{int* } s) \{ \\
  f(p,q); \\
  f(r,s); \\
\}
\end{align*}
\]
Assume p and q may alias

\[
\begin{align*}
\text{void } f(&\text{int* } x, \text{int* } y) \{ \\
&*x = 1; \\
&*y = 2;
\}
\end{align*}
\]

\[
\begin{align*}
\text{void } g(&\text{int* } p, \text{int* } q, \\
&\text{int* } r, \text{int* } s) \{ \\
&f(p, q); \\
&f(r, s);
\}
\end{align*}
\]
void f(int* x, int* y) {
    *x = 1;
    *y = 2;
}

Assume p and q may alias

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

\[
\begin{align*}
f(x, y, A_{xy}, A'_{xy}) \{ \\
A'_{xy} &= \text{store}(A_{xy}, x, 1) \\
A''_{xy} &= \text{store}(A'_{xy}, y, 2) \\
\}
g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs}) \{ \\
f(p, q, A_{pq}, A'_{pq}) \\
f(r, s, A_{rs}, A'_{rs}) \\
\}
\end{align*}
\]
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:**

\[
\begin{align*}
    f(x, y, A_{xy}, A'_{xy}) & \{ \\
    A'_{xy} & = \text{store}(A_{xy}, x, 1) \\
    A''_{xy} & = \text{store}(A'_{xy}, y, 2) \\
    \}
\end{align*}
\]

\[
\begin{align*}
    g(p, q, r, s, A_{pq}, A_{rs}, A'_{pq}, A'_{rs}) & \{ \\
    f(p, q, A_{pq}, A'_{pq}) & \\
    f(r, s, A_{rs}, A'_{rs}) & \\
    \}
\end{align*}
\]

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

```c
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 $\geq 1$ 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 $g \in \mathcal{G}_C$ is a triple $\langle V, E, \sigma \rangle$:

  $V \subseteq \mathcal{C}_C \quad E \subseteq \mathcal{C}_C \times \mathcal{C}_C \quad \sigma : \mathcal{V}_P \mapsto \mathcal{C}_C$

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

  $g \in \mathcal{G}_C \quad \pi : \mathcal{V}_I \mapsto \mathbb{Z} \quad pc \in \mathbb{I}$

• malloc returns a fresh memory object
Concrete Semantics: Assumptions

1. Freed memory is not reused:

```c
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:

```c
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 $G_A$ is a triple $\langle V, E, \sigma \rangle$:
  \[ V \subseteq C_A \quad E \subseteq C_A \times C_A \quad \sigma : \mathcal{V}_P \mapsto C_A \]
  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

Concrete points-to graph

Abstract points-to graph

sequence = true
collapsed = false
size = multiple of 4

sequence = false
collapsed = false
size = 8
Simulation Relation between Graphs

- Sequence = true
- Collapsed = false
- Size = multiple of 4

- Sequence = false
- Collapsed = false
- Size = 8

- 0
- 4
Simulation Relation between Graphs

Sequence = false
Size = 16
Collapsed = false

0 4 8 12

0 4
0 4
0 4
0 4
Null Null Null Null

Sequence = true
Size = multiple of 4

0 4

Sequence = false
Size = false
Size = 8

0 4
Simulation Relation between Graphs

sequence = false  size = 16  collapsed = false
0 4 8 12

sequence = false  collapsed = false  size = 8
0 4
0 4
0 4
0 4

sequence = true  collapsed = false  size = multiple of 4

sequence = false  collapsed = false  size = 8
0 4

$\rho$
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

sequence = false
collapsed = false
size = 8

sequence = true
collapsed = false
size = multiple of 4

0 4 8 12

NULL

0 4
0 4
0 4
0 4

0 4
0 4
0 4
0 4

sequence = false
collapsed = false
size = 8

0 4
Simulation Relation between Graphs

- sequence = false
- size = 16
- collapsed = false

0 4 8 12

- sequence = false
- collapsed = false
- size = 8

0 4 0 4 0 4

- sequence = false
- collapsed = false
- size = 8

NULL NULL NULL NULL

- sequence = true
- collapsed = false
- size = multiple of 4

0 4

- sequence = false
- collapsed = false
- size = 8

0 4

\( \rho \)
Simulation Relation between Graphs

sequence = false  size = 16  collapsed = false

0  4  8  12

sequence = false  collapsed = false  size = 8

0  4
0  4
0  4
0  4

NULL  NULL  NULL  NULL

sequence = true  collapsed = false  size = multiple of 4

sequence = false  collapsed = false  size = 8

0  4
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

0 4 8 12

sequence = false  collapsed = false
size = 8

0 4
NULL

sequence = true  collapsed = false
size = multiple of 4

0 4

sequence = false  collapsed = false
size = 8

0 4
NULL
Simulation Relation between Graphs

sequence = false  size = 16  collapsed = false

0  4  8  12

sequence = false  collapsed = false  size = 8

0  4

0  4

NULL

0  4

0  4

NULL

sequence = true  collapsed = false  size = multiple of 4

0  4

0  4

NULL

NULL

0  4

0  4

NULL

NULL
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

0  4  8  12

sequence = false
collapsed = false
size = 8

0  4
0  4
0  4
0  4

NULL  NULL  NULL  NULL

sequence = true
collapsed = false
size = multiple of 4

sequence = false
collapsed = false
size = 8

0  4

p
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

sequence = false  collapsed = false  size = 8

sequence = true  collapsed = false  size = multiple of 4

sequence = false  collapsed = false  size = 8

NULL  NULL  NULL  NULL
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

0 4 8 12

sequence = true  collapsed = false  size = multiple of 4

0 4

sequence = false  collapsed = false  size = 4

NULL

0 4

0 4

0 4

0 4

ρ
Simulation Relation between Graphs

sequence = false  size = 16
collapsed = false

0  4  8  12

sequence = true  collapsed = false
size = multiple of 4

sequence = false  collapsed = false
size = 4

0  4  0  4  0  4  0  4

NULL  NULL  NULL  NULL

sequence = false  collapsed = false
size = 8

0  4

$p$
Simulation Relation between Graphs

sequence = false  size = 16  collapsed = false

0  4  8  12

sequence = true  collapsed = false  size = multiple of 4

0  4

sequence = false  collapsed = false  size = 4

NULL  NULL  NULL  NULL

ρ
Simulation Relation between Graphs

sequence = false  size = 16  collapsed = false
0  4  8  12

sequence = true  collapsed = false  size = multiple of 4

sequence = false  collapsed = false  size = 4

NULL

ρ
Simulation Relation between Graphs

- $\gamma : \mathcal{G}_A \mapsto 2^{\mathcal{G}_C}$ defined as

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

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

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

- 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 \( (n_2, o_2 - o_1 + o_k) \)

\[
\text{unify}(Y, C) = \text{unify}((N_1, 4), (N_2, 8))
\]
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);
}
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(N,0);
}
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);
}
```c
void g(...) {
    f(p1,p2,p3);
}
void h(...) {
    f(r1,r2,r3);
}
void f(int*q1, int*q2, int*q3) {
    ...
}
```
void g(...) {
    f(p1,p2,p3);
}
void h(...) {
    f(r1,r2,r3);
}
void f(int*q1,int*q2,int*q3) {
    ...
}
void g(...) {
    f(p1,p2,p3);
}

void h(...) {
    f(r1,r2,r3);
}

void f(int*q1,int*q2,int*q3) {
    ...
}
Context-Sensitive Pointer Analysis

```c
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

```c
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?
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 $\rho$ between callee and caller graphs:

1. if $\rho$ is not a function then BU
2. else if $\rho$ is a function but not injective then TD
3. else $\rho$ is an injective function then do nothing
for each function in reverse topological order of the call graph
compute summary

for each callsite
clone callee’s summary into the caller graph and unify formal/actual cells

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

<table>
<thead>
<tr>
<th></th>
<th>#Objects</th>
<th>#Collapsed</th>
<th>Max. Density</th>
<th>% Proven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sea + DSA</td>
<td>258</td>
<td>49%</td>
<td>80%</td>
<td>13</td>
</tr>
<tr>
<td>Sea + our CS</td>
<td>12,789</td>
<td>4%</td>
<td>13%</td>
<td>21</td>
</tr>
</tbody>
</table>
Related Work (1/2)

- Our work is inspired by Data Structure Analysis (DSA) of Lattner et al.: a context and field-sensitive (FS) PA with explicit heap representation:
  - context-sensitivity (CS) cannot be exploited: \textbf{CC} is not guaranteed
  - array-insensitive
- Many Software Model Checkers (e.g., CBMC, Smack, Cascade) are based on static memory partitioning via a FS unification-based pointer analysis
  - they are context-insensitive (CI)
  - Smack and Cascade perform type inference to refine partitions
Deductive verification systems (HAVOC, VCC, and Frama-C):
- More precise memory models
- require quantified axioms
- Frama-C/Jessie is CS but it rejects programs that do not satisfy CC

Pointer analyses combined with other abstractions:
- Miné’s cell-based memory model: CI, flow-sensitive, FS with numerical abstraction of offsets

Shape analysis and its combination with numerical abstractions can infer more expressive invariants but scalability is challenging
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
All Software Publicly Available

- https://github.com/seahorn/sea-dsa
- https://github.com/seahorn/llvm-dsa
- https://github.com/seahorn/seahorn
- https://bitbucket.org/spacer/code
- https://github.com/seahorn/crab-llvm
- https://github.com/seahorn/crab