Fast as a Shadow, Expressive as a Tree: Hybrid Memory Monitoring for C

Nikolai Kosmatov[1]

with Arvid Jakobsson[2], Guillaume Petiot[1] and Julien Signoles[1]

[1]firstname.lastname@cea.fr
[2]arvid.jakobsson@gmail.com

list

SASEFOR, November 24, 2015

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

A brief history

- 90's: CAVEAT, Hoare logic-based tool for C code at CEA
- 2000's: CAVEAT used by Airbus during certification process of the A380 (DO-178 level A qualification)
- 2002: Why and its C front-end Caduceus (at INRIA)
- 2006: Joint project on a successor to CAVEAT and Caduceus
- 2008: First public release of Frama-C (Hydrogen)
- Today: Frama-C Sodium (v.11)
  - Multiple projects around the platform
  - A growing community of users. . .
  - and of developers

Frama-C at a glance

- A Framework for Modular Analysis of C code
- Developed at CEA LIST in collaboration with INRIA Saclay
- Released under LGPL license
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy
- `http://frama-c.com/` [Kirchner et al. FAC 2015]

ACSL: ANSI/ISO C Specification Language

- Based on the notion of contract, like in Eiffel, JML
- Allows users to specify functional properties of programs
- Allows communication between various plugins
- Independent from a particular analysis
- Manual at `http://frama-c.com/acsl`

Basic Components

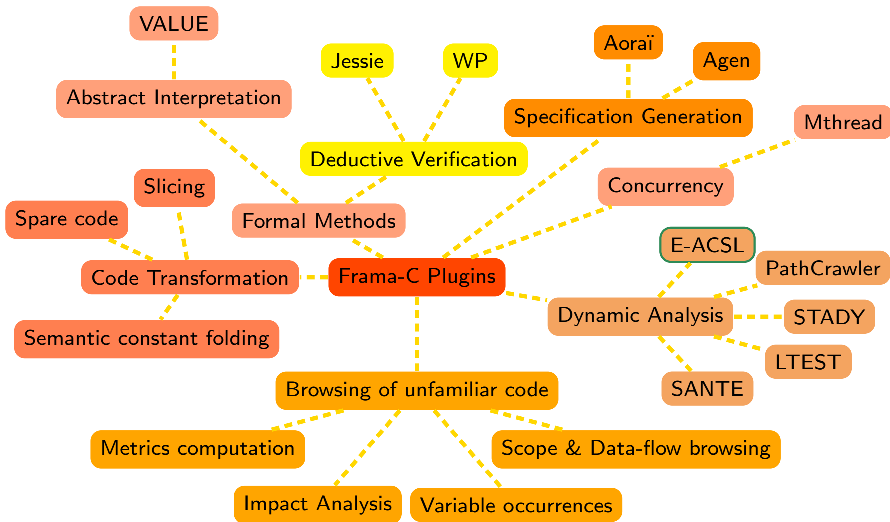
- First-order logic
- Pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-in predicates and logic functions particularly over pointers: `\valid(p) \valid(p+0..2), \separated(p+0..2,q+0..5), \block_length(p)`

Example: a C program annotated in ACSL

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven in Frama-C/WP

Main Frama-C plugins



Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

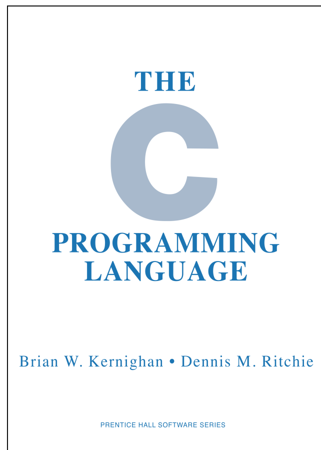
Conclusion and future work

The C language is risky!

- Low-level operations
- Widely used for critical software
- Lack of security mechanisms

Runtime errors are common:

- Division by 0
- Invalid array index
- Invalid pointer
- Non initialized variable
- Out-of-bounds shifting
- Arithmetical overflow
- ...



Memory safety violations in C programs

Spacial safety violations:

- out-of-bounds access, unintialized/manufactured/null pointer access

Temporal safety violations:

- dangling stack/heap pointer access (use-after-free), multiple dealloc's

Many advanced techniques and tools proposed, relying on:

- object metadata [Jones & Kelly, VEE'97], [Ruwase & Lam, NDSS'04], [Xu et al, FSE'04], [Djurjati & Adve, ICSE'06], [Arkitidis et al, USENIX'09]
- fat pointers Safe C [Austin et al, PLDI'94], FailSafe C [Oiwa, PLDI'09]
- combined techniques CAWDOR [SCAM'12], MemSafe [Soft.,Pract.Exp.,2013]
- shadow memory Valgrind [PLDI'07], AddressSanitizer [USENIX ATC'12]

Our objective is different: efficiently support runtime checking of memory related annotations for an expressive specification language

E-ACSL Language

E-ACSL, an executable subset of ACSL:

- it is verifiable in finite time, suitable for runtime assertion checking
- limitations: only bounded quantification, no axioms, no lemmas
- Includes builtin memory-related predicates

E-ACSL2C is a runtime verification tool for E-ACSL specifications:

- it translates annotated program p into another program p'
- p' exits with error message if an annotation is violated
- otherwise p and p' have the same behavior

[Delahaye et al. SAC 2013]

Motivation

Support efficient runtime checking for memory-related E-ACSL constructs for a pointer p such as:

E-ACSL keyword	Its semantics
`\valid(p)`	True iff *p* is valid
`\initialized(p)`	True iff $*p$ has been initialized
`\block_length(p)`	Length of *p*'s memory block
`\base_address(p)`	Base address of *p*'s memory block
`\offset(p)`	Offset of *p* in its memory block

Motivation

Support efficient runtime checking for memory-related E-ACSL constructs for a pointer p such as:

E-ACSL keyword	Its semantics	Monitoring level
`\valid(p)`	True iff p is valid	Byte
`\initialized(p)`	True iff $*p$ has been initialized	Byte
`\block_length(p)`	Length of p's memory block	Block
`\base_address(p)`	Base address of p's memory block	Block
`\offset(p)`	Offset of p in its memory block	Block

Block-level information such as block length and base address is not needed to verify byte-level predicates.

Example C program p annotated with E-ACSL

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;




/*@ assert len > 0 ; */

a_inv = malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */


  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
free(a_inv);
```

Example C program p' translated by E-ACSL2C (simplified)

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;





/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
free(a_inv);
```

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

The memory monitoring library, an overview

E-ACSL2C performs a non-invasive C code instrumentation of p into p':

- p' records memory block (object) metadata *in the store*
  - validity information (including base address, size) whenever a new block is allocated
  - initialization information whenever a byte is assigned
- p' queries the store to evaluate memory related E-ACSL constructs

The memory monitoring library provides primitives for record/query operations.

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: $\{(a, 16)\}$

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
• __store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4)}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4)}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
• __store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4)}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
• a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
• __e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), ~~(a_inv, 16)~~}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
• __delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), (&i, 4), ~~(&a_inv, 4), (a_inv, 16)~~}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
• __delete_block(& i);
__delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), (&len, 4), ~~(&i, 4), (&a_inv, 4), (a_inv, 16)~~}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
• __delete_block(& len);
__delete_block(& a);
```

Memory model: {(a, 16), ~~(&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)~~}

Instrumented program p' with memory monitoring

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(& a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
} // array a_inv is inversed
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
• __delete_block(& a);
```

Memory model: {~~(a, 16), (&len, 4), (&i, 4), (&a_inv, 4), (a_inv, 16)~~}

The problem:

How to store and extract this information efficiently?

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

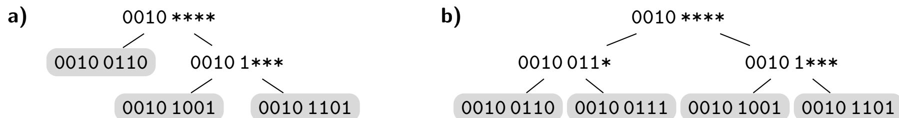
Conclusion and future work

Model 1: Patricia trie

A Patricia trie (compact prefix trie) is an optimized prefix trie, where

- leaves contain stored keys (here, block base addresses)
- any internal node contains greatest common prefix of all its successors

Example of a Patricia trie **a)** before, and **b)** after inserting 0010 0111



Advantages of a Patricia trie: supports byte- and block-level predicates

- sorted structure (e.g. closest predecessor searched for `\base_addr`)
- size of metadata not limited

Disadvantage:

- look-up of metadata in O(k) steps (word length k=32 or 64)

Experiments: comparison to other data structures

Our implementation with a Patricia trie is in average

- 2500 times faster than linked lists
- 200 times faster than unbalanced binary search trees
  - linear worst case complexity !
- 27 times faster than Splay trees (can be 3 times slower or >500 times faster depending on examples)
  - Splay trees move recently accessed elements to the top
  - it pays if frequent successive accesses to the same blocks
  - waste of time if successive accesses to different blocks (ex. big matrix multiplication)

Optimized records and queries in the store

Queries in the store intensively use greatest common prefix computation to decide which branch to follow or which common predecessor to add

Here, the Patricia trie stores binary numbers as keys (base addresses)

- a linear search to compute greatest common prefix can be avoided,
- efficient logarithmic dichotomic search can be used instead

Our greatest common prefix uses dichotomic search optimized by

- bit operations
- pre-computed masks
- pre-computed next step indices (no need for (high+low)/2)

Optimized records and queries in the store, cont'd

Greatest common prefix mask by dichotomic search for 8-bit words:

```
typedef unsigned char byte;
// index             0    1    2    3    4    5    6    7    8
byte  masks[] = {0x00,0x80,0xC0,0xE0,0xF0,0xF8,0xFC,0xFE,0xFF};
int longer [] = {   0,  -1,   3,  -3,   6,  -5,   7,   8,  -8};
int shorter[] = {   0,   0,   1,  -2,   2,  -4,   5,  -6,  -7};
byte gtCommonPrefixMask(byte a, byte b) {
  byte nxor = ~(a ^ b);   // a bit = 1 iff this bit is equal in a and b
  int i = 4;              // search starts in the middle of the word
  while(i > 0)            // if more comparisons needed
    if (nxor >= masks[i])
      i = longer[i];      // if first i bits equal,try a longer prefix
    else i = shorter[i];  // otherwise, try a shorter prefix
  return masks[-i];       // if i<=0, masks[-i] is the answer
}
```

Experiments: optimized greatest common prefix

Our optimized greatest common prefix makes the execution of the instrumented code

- in average 2.7 times faster
- going up to 4.7 times faster on some examples

compared to a linear search optimized only by bit operations

[Kosmatov et al. RV 2013]

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

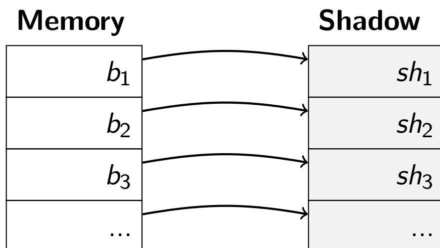
Evaluation

Conclusion and future work

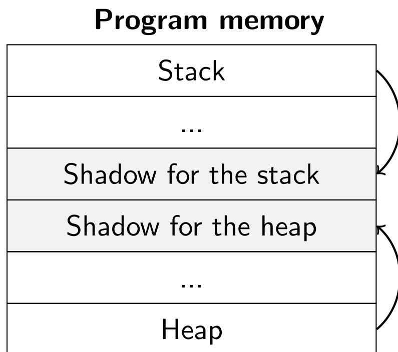
Model 2: Shadow memory

Inspired by the shadow memory technique:

- The shadow memory is a large array of metadata
- Each byte of user memory (allocated or not) corresponds to one element of this array
- Mapping user memory to shadow memory in constant time



Model 2: Shadow memory, continued



Advantage of Shadow memory:

- constant-time lookups of metadata

Disadvantages:

- size of metadata is limited, only supports byte-level predicates
- cannot cover all memory in general (but does for most programs)

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Proposed solution: Hybrid model

Objective: Reconcile the benefits of the tree-based and shadow-memory-based storage.

Idea: Use both models in tandem. Place blocks in one or the other model, depending on:

- the size of the block,
- the metadata needed for that block.

Design principles of the Hybrid model, 1/3

Storage

P1 Metadata of all memory locations that require block-level monitoring should be stored in the Patricia trie store.

P2 For memory locations that require byte-level monitoring, their metadata are typically stored in the shadow memory store.

P3 If a block does not completely belong to the interval of addresses supported by the shadow memory, its metadata should be stored in the Patricia trie store.

P4 For blocks longer than a (parameterizable) constant length C, the metadata should be stored in the Patricia trie store.

Principles of the Hybrid model, 2/3

Evaluation

P5 The evaluation of block-level predicates should always query the Patricia trie.

P6 For predicates that require byte-level monitoring, the evaluation first tries to find the required information in the shadow memory store. If the block is unknown in the shadow mem ry store, it queries the Patricia trie store.

Principles of the Hybrid model, 3/3

Deallocation

P7 The deallocation of an existing block first tries to remove the block metadata from the shadow memory store. If the block is unknown in the shadow memory store, it queries and tries to remove it from the Patricia trie store.

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

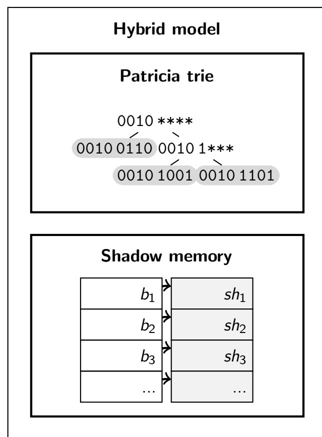
The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Proposed solution: Hybrid model, example



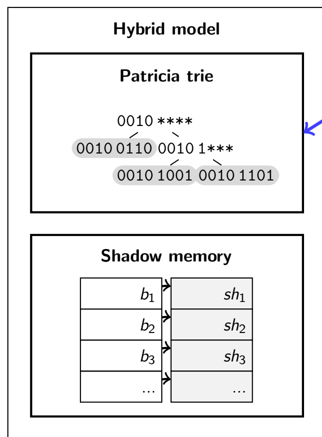

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example



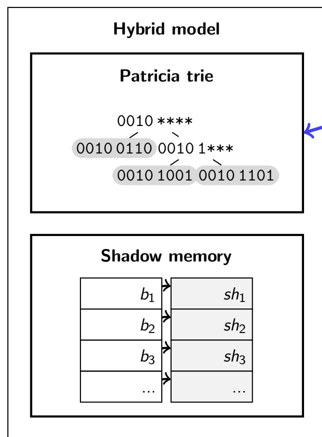

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example



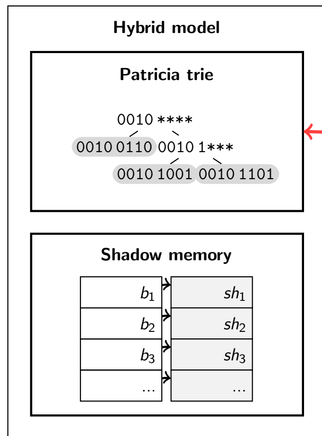

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example



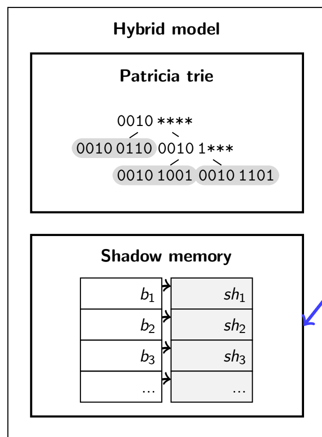

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example



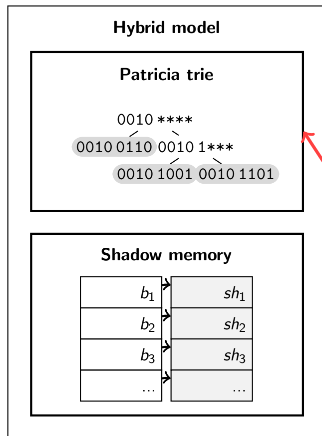

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example



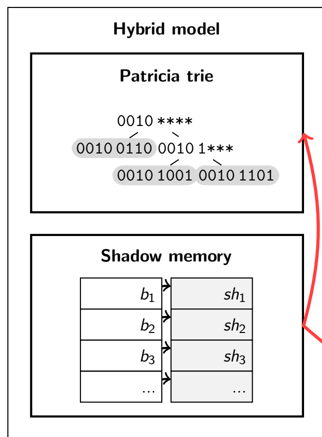

```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;


/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Proposed solution: Hybrid model, example




```
char a[128];
char b;
/*@ assert
  \block_length(b) == 1 ; */
char c;

/*@ assert
  \block_length(a) == 128; */
char *p = &a[0];
/*@ assert \valid(p) ; */
```

Advantages of the Hybrid:

- supports byte- and block-level predicates
- as fast as the shadow, when only byte-level monitoring is needed

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Dataflow analysis

Goal: reduce irrelevant instrumentation

- Backward dataflow analaysis
- Computes an over-approximation
  - of left-values to be monitored
  - and their observation purpose (validity, initialization, . . .)
- Ignores (approximates) offsets
- Parameterized by an alias analysis

[Jakobsson et al. JFLA 2015]

Instrumented program p' after pre-analysis: what is useful?

```
int a[] = {1,2,3,4}, len = 4, i, *a_inv;
__store_block(a,16);
__store_block(& len,4);
__store_block(& i,4);
__store_block(& a_inv,4);
/*@ assert len > 0 ; */
e_acsl_assert(len > 0);
a_inv = __e_acsl_malloc(sizeof(int)*len);
for (i = len - 1; i >= 0; i--) {
  /*@ assert \valid(a + i) ; */
  int __e_acsl_valid = __valid(a + i,sizeof(int));
  e_acsl_assert(__e_acsl_valid);
  a_inv[len - i - 1] = a[i];
}
__e_acsl_free(a_inv);
__delete_block(& a_inv);
__delete_block(& i);
__delete_block(& len);
__delete_block(a);
```

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

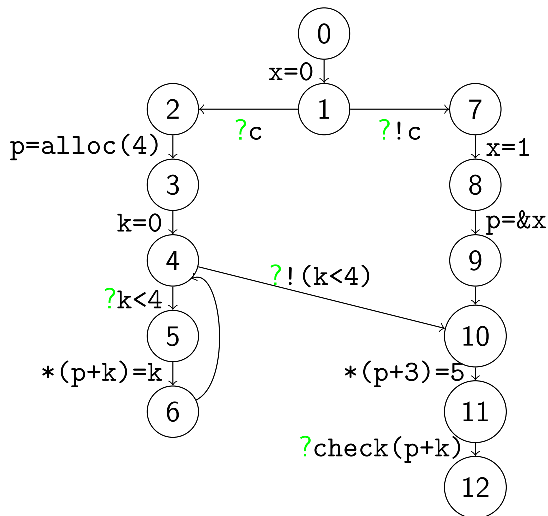
Evaluation

Conclusion and future work

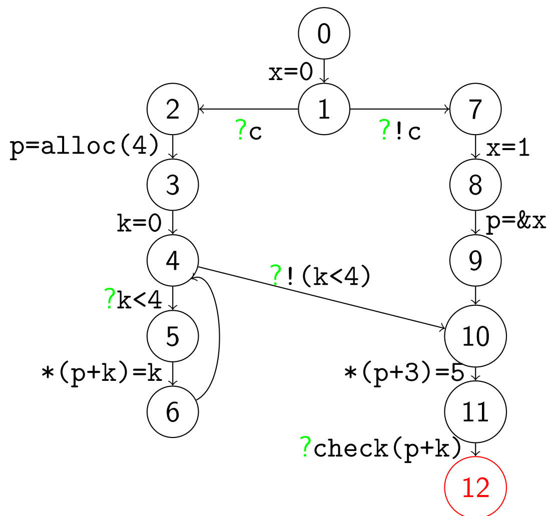
Illustrative example

```
void f(int c) {
  char *p;
  int x = 0;
  if (c) {
    p = (char*)malloc(sizeof(char) * 4);
    for(int k = 0; k < 4; i++)
      *(p+k) = (char)k;
  } else {
    x = 1;
    p = (char*)&x;
  }
  *(p+3) = 5;
/*@ assert ∀ integer k; 0<=k<4 ⟹ \valid(p+k) ∧ \initialized(p+k); */
}
```

Illustrative example, continued

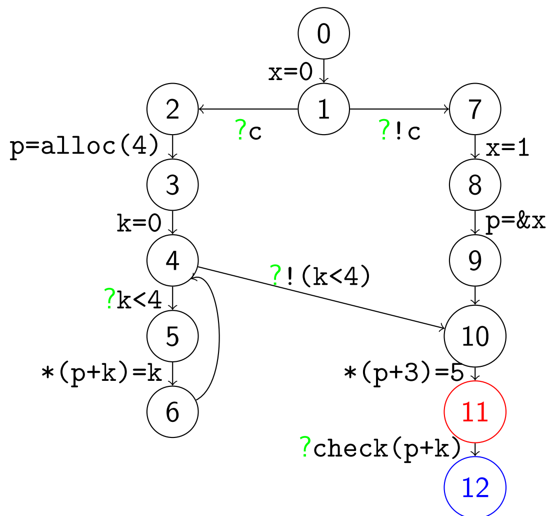


Illustrative example, continued



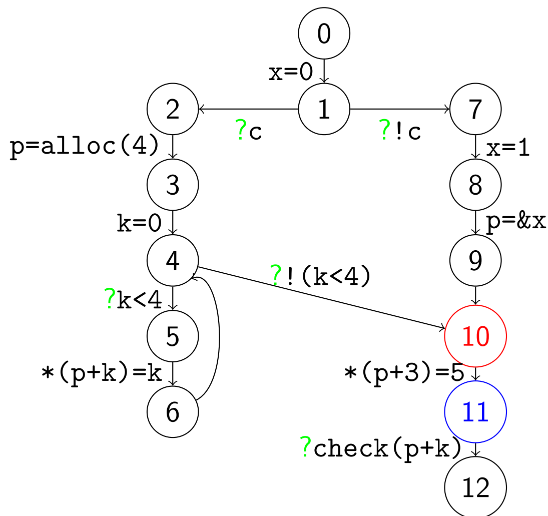
12	$\emptyset$
11	
10	
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

Illustrative example, continued



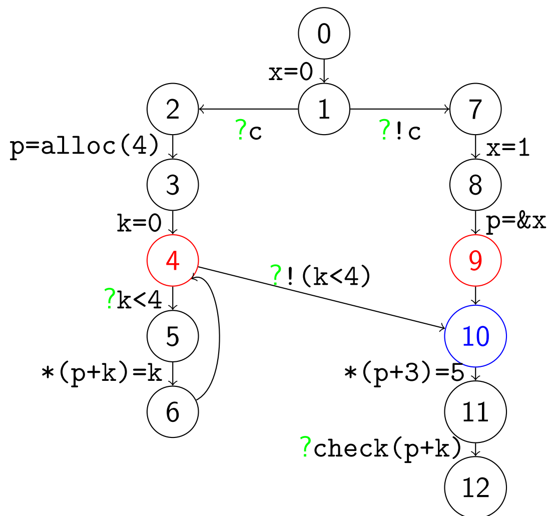
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

Illustrative example, continued



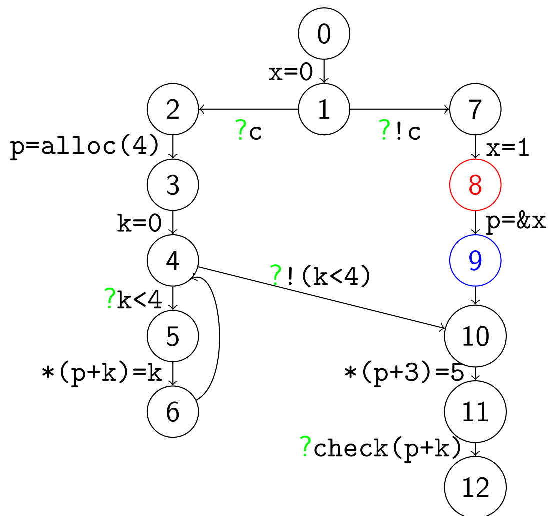
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

Illustrative example, continued



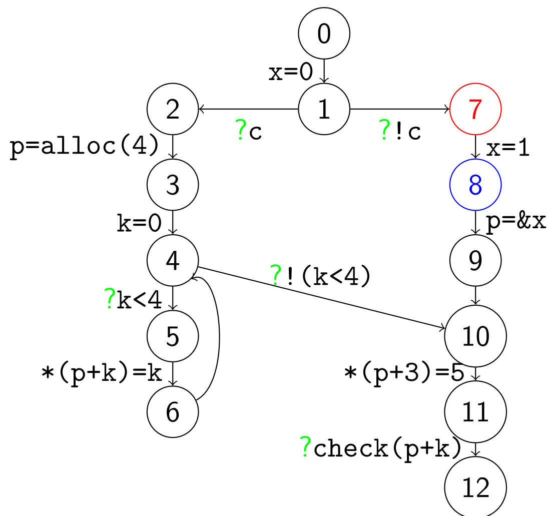
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	
7	
6	
5	
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



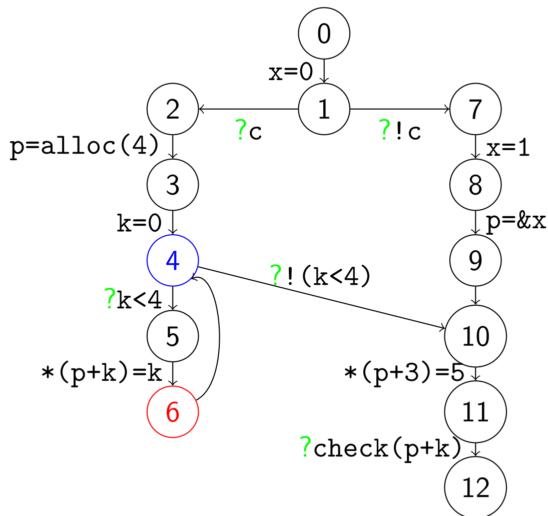
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	
6	
5	
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



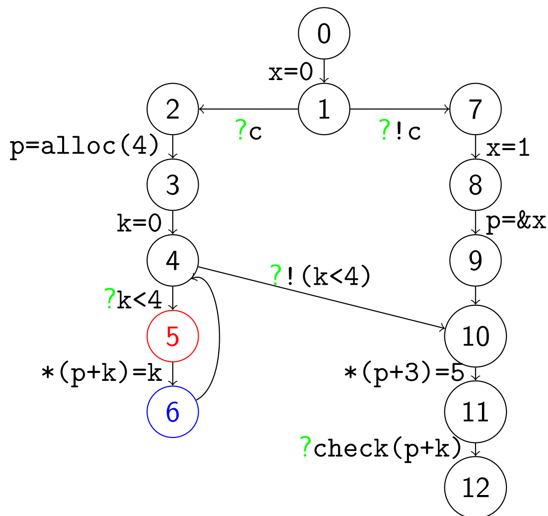
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	
5	
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



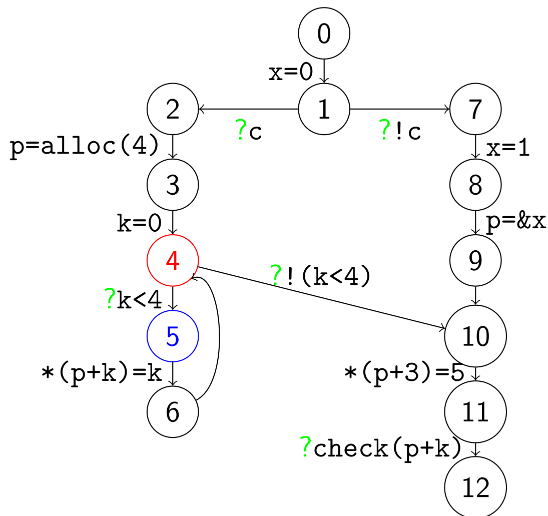
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



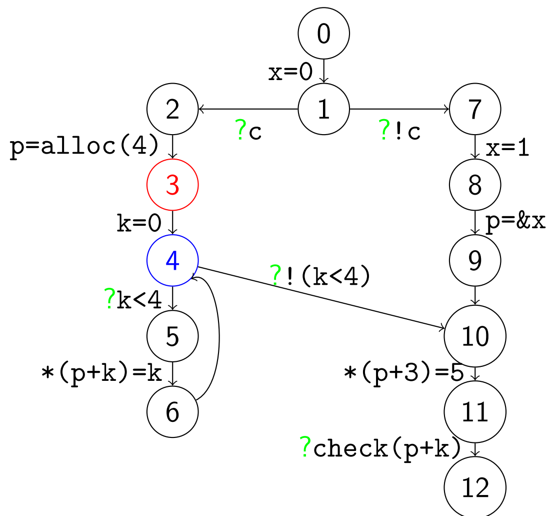
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



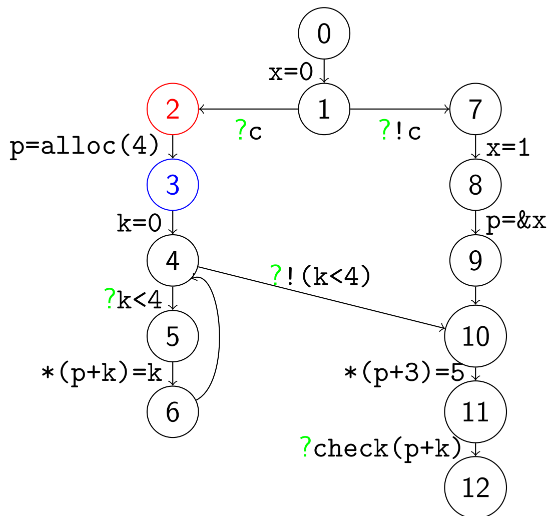
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	
2	
1	
0	

Illustrative example, continued



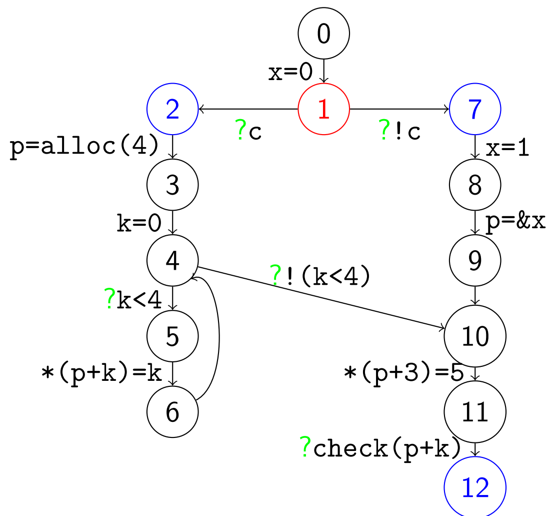
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	$p \mapsto v, i; \&x \mapsto i$
2	
1	
0	

Illustrative example, continued



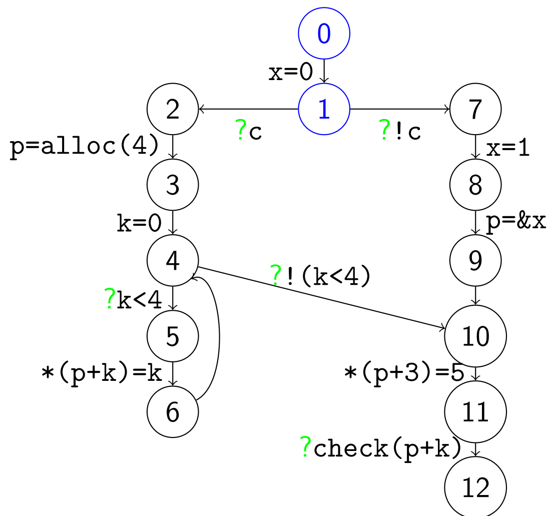
12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	$p \mapsto v, i; \&x \mapsto i$
2	$p \mapsto i; \&x \mapsto i$
1	
0	

Illustrative example, continued



12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	$p \mapsto v, i; \&x \mapsto i$
2	$p \mapsto i; \&x \mapsto i$
1	$p \mapsto i; \&x \mapsto i$
0	

Illustrative example, continued



12	$\emptyset$
11	$p \mapsto v, i; \&x \mapsto i$
10	$p \mapsto v, i; \&x \mapsto i$
9	$p \mapsto v, i; \&x \mapsto i$
8	$p \mapsto i; \&x \mapsto i$
7	$p \mapsto i$
6	$p \mapsto v, i; \&x \mapsto i$
5	$p \mapsto v, i; \&x \mapsto i$
4	$p \mapsto v, i; \&x \mapsto i$
3	$p \mapsto v, i; \&x \mapsto i$
2	$p \mapsto i; \&x \mapsto i$
1	$p \mapsto i; \&x \mapsto i$
0	$p \mapsto i$

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Evaluation

Research questions:

- **RQ1:** How does the hybrid model compare to the tree-based model?
- **RQ2:** How does the hybrid model compare to the shadow-memory-based model?
- **RQ3:** How does memory monitoring for E-ACSL, using the hybrid model, compare to Valgrind?
- **RQ4:** What are the benefits of the pre-analysis?

Benchmark results

Execution time (in sec.) for the hybrid memory store (H) w.r.t. the Patricia trie store (PT), the shadow memory store (Sh) and Valgrind

Example	Orig.	PT	Sh	H	H vs PT	H vs Sh	Valgrind
bubbleSort	0.56	4.13	4.82	4.50	+8.96%	−6.64%	9.47
binSearch	0.00	2.85	6.68	2.90	+1.75%	−56.59%	0.48
mergeSort	0.04	86.28	0.43	0.42	−99.51%	−2.33%	2.68
quickSort	0.00	1.15	0.06	0.07	−93.91%	+16.67%	0.54
RedBlTree	0.03	0.59	0.19	0.28	−52.54%	+47.37%	2.29
merge	0.01	1.25	0.08	0.08	−93.60%	0.00%	1.08
matrixMult	0.13	3.46	0.67	0.68	−80.35%	+1.49%	1.85
matrixInv	0.01	3.45	1.59	1.66	−51.88%	+4.40%	0.70
insertSort	2.67	2.78	2.80	2.78	0.00%	−0.71%	35.61
Total	3.47	105.94	17.32	13.37	−87.38%	−22.81%	54.7

- **Hybrid vs. Patricia trie:** hybrid is faster, and as expressive
- **Hybrid vs. Shadow memory:** hybrid is as fast, and more expressive
- **Hybrid vs. Valgrind:** execution time for the two is not comparable, but memory usage of Valgrind is higher
- **The pre-analysis:** provides a speedup of 73% for Hybrid model

Outline

Context and motivation
- Frama-C, a platform for analysis of C code
- Motivation

The memory monitoring library
- An overview
- Patricia trie model
- Shadow memory based model

The Hybrid model
- Design principles
- Illustrating example

Dataflow analysis
- An overview
- How it proceeds

Evaluation

Conclusion and future work

Conclusion

Efficient memory monitoring with hybrid memory model for runtime checking with Frama-C:

- applicable to an expressive specification language E-ACSL including memory related constructs
- the hybrid memory model reconciles the expressivity of the tree-based model with the efficiency of the shadow-memory based
- further optimized by static analysis

Future work

- Extend the pre-analysis to the monitoring-level (block / byte) per variable
- Integrate the complete solution into the released version
- Investigate other (static / dynamic) heuristics for choosing the store to be used in the hybrid model
- Better support of some kinds or errors

Thank you!

`http://frama-c.com/eacsl.html`