A Memory Model and Other Extensions of ACSL for Deductive Verification of Linux Kernel Modules

User-guided separation analysis
Operations on bounded integers
Proofs as C code

Mikhail Mandrykin, Alexey Khoroshilov

ISP RAS

LRI, May 29th, 2017

Contents

Basic region-based model

From the paper

T. Hubert and C. Marché, "Separation analysis for deductive verification," in Heap Analysis and Verification (HAV'07), (Braga, Portugal), pp. 81–93, March 2007

Basic region-based model

Sample program

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  void swap(struct s **x,
5            struct s **y)
6  {
7    struct s *t = *x;
8    *x = *y;
9    *y = t;
10   struct pair *p = t->p;
11   t->p = (*x)->p;
12   (*x)->p = p;
13 }
```

```
14 void caller()
15 {
16   struct s x, y, z;
17   struct s *px = &x,
18            *py = &y,
19            *pz = &z;
20   // ...
21   swap(&px, &py);
22   swap(&py, &pz);
23 }
```

Basic region-based model

Regions

Definition

Regions are sets of (structurally distinct) typed pointer expressions s. t. any pair of pointer expressions taken from **different** regions necessarily addresses **disjoint** memory areas (w.r.t the corresponding pointer types)

$$\forall r_1, r_2 \in \mathbb{R}.\ r_1 \neq r_2 \implies \\ \forall p_1 \in r_1, p_2 \in r_2. \\ ((\texttt{char *})p_1)\big[0 \ldots \texttt{sizeof}(p_1)\big] \cap \\ ((\texttt{char *})p_2)\big[0 \ldots \texttt{sizeof}(p_2)\big] = \emptyset$$

Basic region-based model

Preliminary transformations (before)

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  void swap(struct s **x,
5            struct s **y)
6  {
7    struct s *t = *x;
8    *x = *y;
9    *y = t;
10   struct pair *p = t->p;
11   t->p = (*x)->p;
12   (*x)->p = p;
13 }
```

```
14 void caller()
15 {
16   struct s x, y, z;
17   struct s *px = &x,
18            *py = &y,
19            *pz = &z;
20   // ...
21   swap(&px, &py);
22   swap(&py, &pz);
23 }
```

Basic region-based model

Preliminary transformations (after)

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  struct sP { struct s* sM; };
5  void swap(struct sP *x,
6            struct sP *y)
7  {
8    struct s *t = x->sM;
9    x->sM = y->sM;
10   y->sM = t;
11   struct pair *p = t->p;
12   t->p = x->sM->p;
13   x->sM->p = p;
14 }
```

```
15 void caller()
16 {
17   struct s *x = alloca(...),
18            *y = alloca(...),
19            *z = alloca(...);
20   struct sP *px = alloca(...),
21             *py = alloca(...),
22             *pz = alloca(...);
23   px->sM = x;
24   py->sM = y;
25   pz->sM = z;
26   // ...
27   swap(px, py);
28   swap(py, pz);
29 }
```

Basic region-based model

Region analysis. Initial separation. Parameter & local regions

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  struct sP { struct s* sM; };
5  void swap(struct sP *x,
6            struct sP *y)
7  {
8    struct s *t = x->sM;
9    x->sM = y->sM;
10   y->sM = t;
11   struct pair *p = t->p;
12   t->p = x->sM->p;
13   x->sM->p = p;
14 }
```

```
15 void caller()
16 {
17   struct s *x = alloca(...),
18            *y = alloca(...),
19            *z = alloca(...);
20   struct sP *px = alloca(...),
21             *py = alloca(...),
22             *pz = alloca(...);
23   px->sM = x;
24   py->sM = y;
25   pz->sM = z;
26   // ...
27   swap(px, py);
28   swap(py, pz);
29 }
```

Basic region-based model

Functional consistency

Assume existence of a map (function) Δ from arbitrary pairs (r, f), where r — region, f — pointer field, to regions. Assign every expression of the form p->f region $\Delta(r(p), f)$, where $r(p)$ — region assigned to p. Then we have functional consistency

$$\forall r_1, r_2 \in \mathbb{R}, f \in \mathbb{F}_p \,.\, r_1 = r_2 \implies \Delta(r_1, f) = \Delta(r_2, f)$$

Unification

Unify (equate) regions on

- pointer assignments (p_1 = $p_2 \implies r(p_1) = r(p_2)$),
- pointer comparison operations ($p_1 \lozenge p_2 \implies r(p_1) = r(p_2)$),
- pointer arithmetic/array indexing ($p + i \implies r(p + i) = r(p)$)

Basic region-based model

Region analysis. Functional consistency & unification

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  struct sP { struct s* sM; };
5  void swap(struct sP *x,
6            struct sP *y)
7  {
8    struct s *t = x->sM;
9    x->sM = y->sM;
10   y->sM = t;
11   struct pair *p = t->p;
12   t->p = x->sM->p;
13   x->sM->p = p;
14 }
```

```
15 void caller()
16 {
17   struct s *x = alloca(...),
18            *y = alloca(...),
19            *z = alloca(...);
20   struct sP *px = alloca(...),
21             *py = alloca(...),
22             *pz = alloca(...);
23   px->sM = x;
24   py->sM = y;
25   pz->sM = z;
26   // ...
27   swap(px, py);
28   swap(py, pz);
29 }
```

Basic region-based model

Polymorphic regions & equality constraint propagation

Parameter regions are treated as **polymorphic**. Equality constraints are propagated from the callee to the caller. So we can assume the existence of call site association maps $\Sigma_{f@l} : \mathbb{R} \to \mathbb{R}$, $\Sigma_{f@l}(r_{callee}) = r_{caller}$. Hence if

$$\Sigma_{f@l}(r_{callee1}) = r_{caller1},$$
$$\Sigma_{f@l}(r_{callee2}) = r_{caller2},$$

then

$$r_{callee1} = r_{callee2} \implies$$
$$\Sigma_{f@l}(r_{callee1}) = \Sigma_{f@l}(r_{callee2}) \implies$$
$$r_{caller1} = r_{caller2}$$

New caller regions may be added during propagation

Basic region-based model

Equality propagation & call site association maps

```
1  struct pair { int a, b; };
2  struct s { struct pair *p;
3             int v };
4  struct sP { struct s* sM; };
5  void swap(struct sP *x,
6            struct sP *y)
7  {
8    struct s *t = x->sM;
9    x->sM = y->sM;
10   y->sM = t;
11   struct pair *p = t->p;
12   t->p = x->sM->p;
13   x->sM->p = p;
14 }
```

```
15 void caller()
16 {
17   struct s *x = alloca(...),
18            *y = alloca(...),
19            *z = alloca(...);
20   struct sP *px = alloca(...),
21             *py = alloca(...),
22             *pz = alloca(...);
23   px->sM = x;
24   py->sM = y;
25   pz->sM = z;
26   // ...
27   swap(px, py);
28   // Σ_swap@27={·↦·,·↦·,·↦·,·↦·}
29   swap(py, pz);
30   // Σ_swap@29={·↦·,·↦·,·↦·,·↦·}
31 }
```

Contents

Limitations of basic region-based model

Brief enumeration

- Prefix pointer type casts
```
struct ext { struct pref p; ... } e;
struct pref *p = (struct pref*)&e
```
- Discriminated unions
```
union u { int a; char b; };
```
- Nested structures & arrays
```
struct outer { ...; struct inner i; ... } o;
```
- Non-prefix pointer type casts / non-discriminated unions
```
int a; char *pc = (char *)&a;
```
- Pointer arithmetic beyond array indexing
```
struct outer * po = container_of(pi, struct outer, inner);
```
- Call site region split
```
void swap(int *a, int *b) { ... } ... swap(a[0], a[1]);
```
- Redundant unification
```
int *p = &a; handle(p); p = &b; handle(p);
```

Limitations of basic region-based model

Prefix pointer type casts

```
struct prefix { int a; };
struct ext { struct prefix p; int b; };
//...
struct ext *e = malloc(sizeof(struct ext));
struct prefix *p = (struct prefix *)e;
p->a = 0;
e = (struct ext*)p;
e->p.a = 1;
```

Limitations of basic region-based model

Prefix pointer type casts as hierarchical casts

- Inline prefix (base) structures

```
struct prefix { int a; };
struct ext { struct prefix p; int b; };
e->p.a = 1;
```

$\Rightarrow$

```
struct prefix { int a; };
struct ext { int a; int b; };
e->a = 1;
```

- Ignore prefix casts in region analysis (use the same region)

```
struct prefix * p = (struct prefix *) e;
p->a = 0;  e = (struct ext*) p;  e->a = 1;
```

- Introduce ghost map of **tags** ($\mathbb{T}$) for each region ($T_r : \mathbb{P} \to \mathbb{T}$)
- Introduce partial order relation $\preceq$ on tags s. t.
  $\forall t_1, t_2 \in \mathbb{T}.\ t_1 \preceq t_2 \iff t_2$ is prefix of t_1, $\therefore$ $\texttt{ext} \preceq \texttt{prefix}$
- Assign exact tag to T_r on every allocation

```
struct ext *e = malloc(sizeof(struct ext)); // T_e[e] ← ext
```

- For every cast $(t_2)p$, where $t_1\ *p, t_2 \preceq t_1$, check $T_r[p] \preceq t_2$

```
e = (struct ext*)p; // T_e[p] ⪯ ext?
```

Limitations of basic region-based model

Handling prefix pointer type casts

```
struct prefix { int a; };
struct ext { int a; int b; };
//...
struct ext *e = malloc(sizeof(struct ext));
// T_e[e] ← ext
struct prefix *p = (struct prefix *)e;
p->a = 0;
e = (struct ext*)p;
// T_e[p] ⪯ ext?
e->a = 1;
```

Limitations of basic region-based model

Discriminated unions

Definition

Moderated union A **dynamically allocated** union is moderated iff its underlying memory is only accessed through a pointer to the corresponding union type

```
union u { int a; char c; } *pu;
```

✓ `pu->a`

✗ `int *pa = &pu->a; ... *pa`

✗ `int *pa = (int *)pu; ... *pa`

Definition

Discriminated union A **moderated** union is discriminated iff each read from a field of this union is preceded by the latest write to the same field

✓ `pu->a = 0; ... if (pu->a) pu->c = 0; ... putc(pu->c);`

✗ `pu->a = 0; ... putc(pu->c);`

Limitations of basic region-based model

Real Linux kernel code fragments

```
static int gelic_wl_set_auth(..., union iwreq_data *data, ...) {
        struct iw_param *param = &data->param; // ✗
// ...
static void igb_dump(...) {
        struct my_u0 { u64 a; u64 b; } *u0;
        union e1000_adv_rx_desc *rx_desc;
        // ...
        u0 = (struct my_u0 *)rx_desc; // ✗
// ...
union nf_inet_addr {
        __u32                   all[4];
        __be32                   ip;
        // ...
};
static inline void ip_vs_addr_set(int af, union nf_inet_addr *dst,
                                  const union nf_inet_addr *src) {
        // ...
        dst->ip = src->ip;
        dst->all[1] = 0; // ✗, last written field is ip
        dst->all[2] = 0;
        dst->all[3] = 0;
}
```

Limitations of basic region-based model

Nested structures rewriting

```
struct inner { int a, b; };
struct outer {
  int c;
  struct inner inner; };
struct outer *o =
  malloc(sizeof(struct outer));

o->inner.a = 0;

free(o);
```

$\Rightarrow$

```
struct inner { int a, b; };
struct outer {
  int c;
  struct inner *inner; };
struct outer *o =
  malloc(sizeof(struct outer));
o->inner =
  malloc(sizeof(struct inner));
o->inner->a = 0;
free(o->inner);
free(o);
```

- Inefficient encoding of disjointness
  ($\backslash$separated(o_1,o_2) $\Longrightarrow$ $\backslash$separated(o_1->inner,o_2->inner))
- Hard to support nested structures as union fields
- Tricky error-prone transformation with complicated semantics

Contents

User-guided separation analysis

Non-hierarchical casts/reinterpretation. Basic trade-off

```
int n;
n = 0;
char *pc = (char *)&n, c;
c = *pc; // ✗ Not previously written as char
n = 1;
```

Soundness

- Functional consistency (->) and unification ensure **partitioning**
- To preserve partitioning for **non-discriminated** access, we should **unify regions** of **different types**

Efficiency

- **Separation** between different types enables **efficient** encoding
- **Unrestricted** cross-type aliasing imposes **inefficient** byte-level encoding

Completeness

- Basic model is **sound** and **efficient** for a **subset** of C
- **Restrict** supported subset of C $\Longrightarrow$ **cannot verify Linux kernel code**

User-guided separation analysis

Typed model with reinterpretation (VCC2)

```
int n;                    // T = {([[&n]], int)}
//                  ([[&n]], int) ∈ T = {([[&n]], int)}? ✓
n = 0;                    // T = {([[&n]], int)}
char *pc =
  (char *)&n, c;          // T = {([[&n]], int)}
//                  ([[&n]], int) ∈ T = {([[&n]], int)}? ✓
//@ split( &n );          // T = {([[&n]] + i, char) | 0≤i<sizeof(int)}
//                  ([[pc]], char) ∈ T = {([[&n]] + i, char) | 0≤i<sizeof(int)}? ✓
c = * pc ;                // T = {([[&n]] + i, char) | 0≤i<sizeof(int)}
//                  ([[pc]], char) ∈ T = {([[&n]] + i, char) | 0≤i<sizeof(int)}? ✓
//@ join( pc );           // T = {([[&n]], int)}
//                  ([[&n]], int) ∈ T = {([[&n]], int)}? ✓
n = 1;                    // T = {([[&n]], int)}
```

User-guided separation analysis

Typed model with reinterpretation (VCC2)

```
int n;
n = 0;
char *pc = (char *)&n, c;
//@ split(&n);
c = *pc;
//@ join(pc);
n = 1;
```

User-guided separation analysis

Integrating memory safety checks with reinterpretation

```
int *pn;                          // T = ∅
pn = malloc(...);                 // T = {([[pn]], int)}
//                                ([[pn]], int) ∈ T = {([[pn]], int)}? ✓
* pn  = 0;                        // T = {([[pn]], int)}
char *pc = (char *)pn, c;         // T = {([[pn]], int)}
//                                ([[&n]], int) ∈ T = {([[pn]], int)}? ✓
//@ split( pn );                  // T = {([[pn]] + i, char) | 0≤i<sizeof(int)}
//                                ([[pc]], char) ∈ T = {([[pc]] + i, char) | 0≤i<sizeof(int)}? ✓
c = * pc ;                        // T = {([[pc]] + i, char) | 0≤i<sizeof(int)}
//                                ([[pc]], char) ∈ T = {([[pc]] + i, char) | 0≤i<sizeof(int)}? ✓
//@ join( pc );                   // T = {([[pn]], int)}
//                                ([[pn]], int) ∈ T = {([[pn]], int)}? ✓
* pn  = 1;                        // T = {([[pn]], int)}
//                                ([[pn]], int) ∈ T = {([[pn]], int)}? ✓ (arrays?!)
free( pn );                       // T = ∅
//                                T = ∅? ✓
```

User-guided separation analysis

A posteriori check for region analysis (assuming **one type per region!**)

```
int * pn ;                          // T = ∅
pn = malloc(...);                   // T = {([[pn]], r1)}
//                                  ([[pn]], r1) ∈ T = {([[pn]], r1)}? ✓
* pn = 0;                           // T = {([[pn]], r1)}
char * pc = (char *)pn, c;          // T = {([[pn]], r1)}
//                                  ([[pn]], r1) ∈ T = {([[pn]], r1)}? ✓
//@ split( pn );                    // T = {([[pn]] + i, r3) | 0≤i<sizeof(int)}
//                                  ([[pc]], r2) ∈ T = {([[pn]] + i, r3) | 0≤i<sizeof(int)}? ✗
c = * pc ;                          // T = {([[pn]] + i, r3) | 0≤i<sizeof(int)}
//                                  ([[pc]], r2) ∈ T = {([[pn]] + i, r3) | 0≤i<sizeof(int)}? ✗
//@ join( pc );                     // T = {([[pn]], r1)}
//                                  ([[pn]], r1) ∈ T = {([[pn]], r1)}? ✓
* pn = 1;                           // T = {([[pn]], r1)}
//                                  ([[pn]], r1) ∈ T = {([[pn]], r1)}? ✓ (arrays?!)
free( pn );                         // T = ∅
//                                  T = ∅? ✓
```

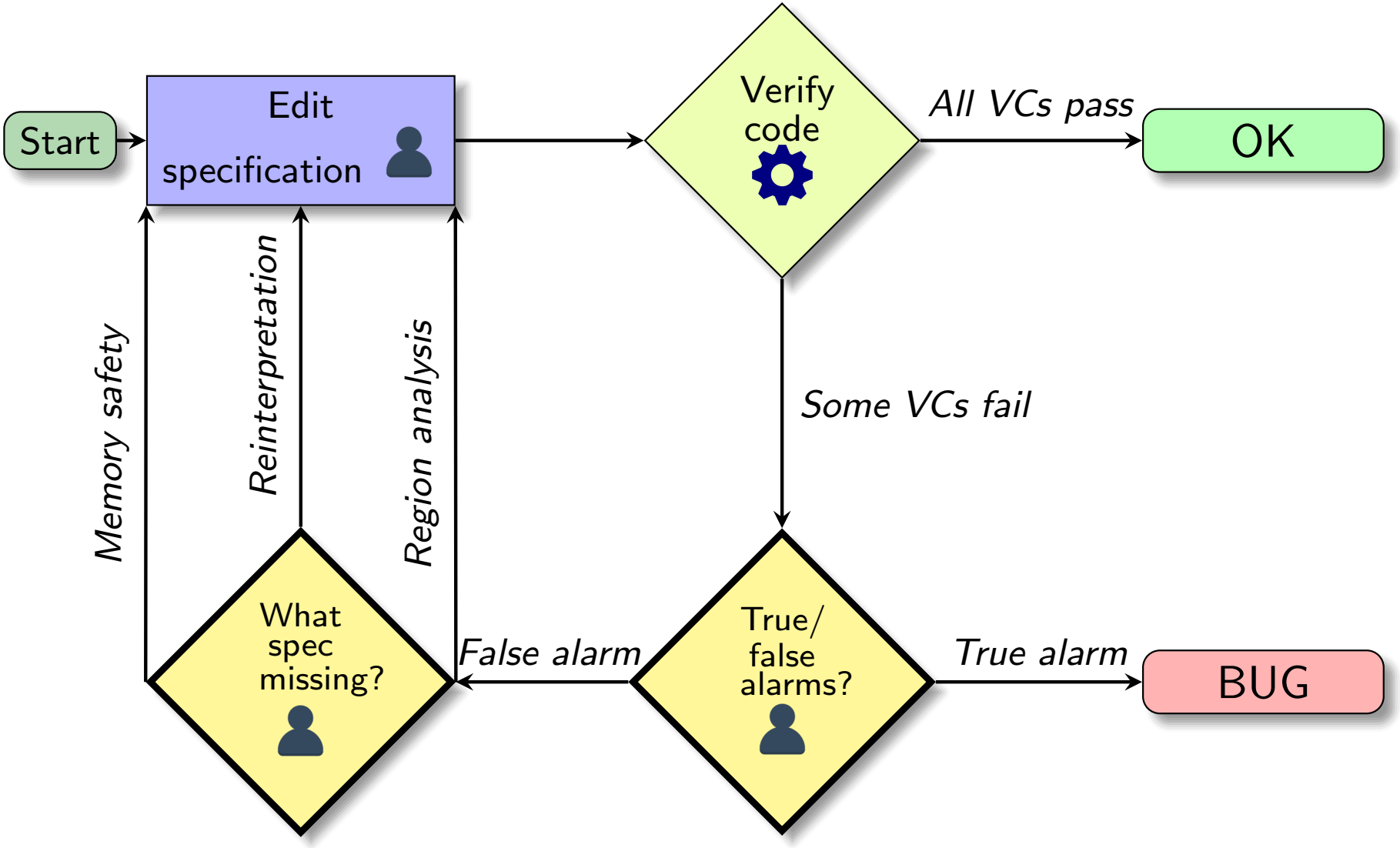
User-guided separation analysis

Proposal

- Use arbitrary (possibly **unsound**) region analysis, assigning exactly one structure tag ($\mathbb{T}$) per region
  $\exists t : \mathbb{R} \to \mathbb{T}.\ \forall r_1, r_2 \in \mathbb{R}.\ r_1 = r_2 \implies t(r_1) = t(r_2)$
- Generate VCs for **simultaneous** check of
  - Correct reinterpretation
  - Memory safety
  - Soundness of region analysis
- Provide **additional specification constructs** to perform reinterpretation and guide region analysis

User-guided separation analysis

User-guided separation & memory safety analysis workflow



User-guided separation analysis

Drawbacks

- **Memory safety**, **reinterpretation** and **region analysis** alarms are combined, **not distinguished** by the tool $\Longrightarrow$
  - Region analysis has to remain **simple** and **predictable**
- Memory model invariants are **inconsistent** with common **C** and **ACSL semantics**, the user can accidentally break the invariants in preconditions
  - Invariants should be easily understandable by the user (*no memory should be valid in more than one region*)
- **Soundness** is not obvious (no region partitioning), relies on formalization of `malloc`/`free`/`split`/`join` maintaining the invariants $\Longrightarrow$
  - Formalize an invariant between **reference** and **model** semantics, prove **preservation**

Reference semantics

Consistency invariants

- **Simplifying assumption**: one `int` size equal to pointer size (`intnat`)
- Invariants:
  - $\forall a, i \in \mathbb{B}^d .\ i < \mathcal{L}[a] \implies \mathcal{V}[a+i]$ $\quad (\mathcal{L} \mapsto \mathcal{V})$
  - $\forall a, i \in \mathbb{B}^d .\ 0 < i < \mathcal{L}[a] \implies \mathcal{L}[a+i] = 0$ $\quad (\mathcal{L} \mapsto \mathcal{L})$
  - $\forall a \in \mathbb{B}^d .\ \mathcal{V}[a] \implies \exists i \in \mathbb{B}^d .\ \mathcal{L}[a-i] > i$ $\quad (\mathcal{V} \mapsto \mathcal{L})$
  - $\neg \mathcal{V}[0]$ $\quad (\texttt{NULL})$

Example

```
struct outer { int a; struct innter { int b, c; }; int d }
  outs[2] = {{ 1, { 20, 30 }, 4 }, { 5, { 60, 70 }, 8 }};
```

$\mathcal{B}$		$\cdots$	1	20	30	4	5	60	70	8	$\cdots$	
$\mathcal{V}$	$\bot$	$\cdots$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\cdots$	
$\mathcal{L}$	0	$\cdots$	8	0	0	0	0	0	0	0	$\cdots$	
	0		`outs`									2^d

Model semantics

Example

```
struct outer { int a; struct innter { int b, c; }; int d }
  outs[2] = {{ 1, { 20, 30 }, 4 }, { 5, { 60, 70 }, 8 }};
```

$\mathcal{M}_{\mathtt{a}}^{r_{\mathtt{outer}}}$		$\cdots$	1				5				$\cdots$	
$\mathcal{M}_{\mathtt{d}}^{r_{\mathtt{outer}}}$		$\cdots$	4				8				$\cdots$	
$\mathcal{M}_{\mathcal{V}}^{r_{\mathtt{outer}}}$	$\bot$	$\cdots$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\cdots$	
$\mathcal{M}_{\mathcal{L}}^{r_{\mathtt{outer}}}$	0	$\cdots$	2	0	0	0	0	0	0	0	$\cdots$	
$\mathcal{M}_{\mathtt{b}}^{r_{\mathtt{inner}}}$		$\cdots$		20				60			$\cdots$	
$\mathcal{M}_{\mathtt{c}}^{r_{\mathtt{inner}}}$		$\cdots$		30				70			$\cdots$	
$\mathcal{M}_{\mathcal{V}}^{r_{\mathtt{inner}}}$	$\bot$	$\cdots$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\cdots$	
$\mathcal{M}_{\mathcal{L}}^{r_{\mathtt{inner}}}$	0	$\cdots$	0	0	0	0	0	0	0	0	$\cdots$	
	0		outs									2^d

Model semantics

Consistency invariants

- $\forall a, i \in \mathbb{B}^d, r \in \mathbb{R}.\ i < \mathcal{M}^r_{\mathcal{L}}[a] \times \texttt{sizeof}(r) \implies \mathcal{V}alid(a + i)$ $\quad (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{V}})$
- $\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\ \mathcal{M}^r_{\mathcal{L}}[a] > 0 \implies uniq\mathcal{L}(a, r, \mathcal{M}^r_{\mathcal{L}}[a])$ $\quad (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{L}})$
- $\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\ \mathcal{M}^r_{\mathcal{V}}[a] \implies$
  $\exists i \in \mathbb{B}^d, r' \in \mathbb{R}.\ \mathcal{M}^{r'}_{\mathcal{L}}[a - i] \times \texttt{sizeof}(r') > i$ $\quad (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{L}})$
- $\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\ \mathcal{M}^r_{\mathcal{V}}[a] \implies uniq\mathcal{V}(a, r)$ $\quad (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{V}})$

$\neg\mathcal{V}alid(0)$ $\quad (\texttt{NULL}^r)$,

where

$$
\begin{aligned}
\mathcal{V}alid(a) &\equiv \exists r \in \mathbb{R}.\ \mathcal{V}alid^r(a, r), \\
\mathcal{V}alid^r(a, r) &\equiv \exists f \in \mathbb{F}(r).\ \mathcal{M}^r_{\mathcal{V}}\big[a - \texttt{offsetof}(f)\big] \\
uniq\mathcal{L}(a, r, l) &\equiv \\
&\quad \forall i \in \mathbb{B}^d, r' \in \mathbb{R}.\ i < l \times \texttt{sizeof}(r) \wedge (r' \neq r \vee i > 0) \implies \mathcal{M}^{r'}_{\mathcal{L}}[a + i] = 0 \\
uniq\mathcal{V}(a, r) &\equiv \Big(\forall f \in \mathbb{F}(r), r' \in \mathbb{R}.\ r' \neq r \Rightarrow \neg\mathcal{V}alid^r\big(a + \texttt{offsetof}(f), r'\big)\Big) \wedge \\
&\quad \big(\forall i \in \mathbb{B}^d.\ 0 < i < \texttt{sizeof}(r) \implies \neg\mathcal{M}^r_{\mathcal{V}}[a + i]\big)
\end{aligned}
$$

Model semantics

Correspondence invariants

$$
\begin{array}{ll}
\big(\forall a \in \mathbb{B}^d.\, \mathcal{V}[a] \implies \mathcal{V}alid(a)\big) \wedge & (\mathcal{V} \mapsto \mathcal{M}_{\mathcal{V}}) \\
\big(\forall a \in \mathbb{B}^d.\, \mathcal{L}[a] > 0 \implies \exists r \in \mathbb{R}.\, \mathcal{L}[a] = \mathcal{M}^r_{\mathcal{L}}[a] \times \texttt{sizeof}(r)\big) \wedge & (\mathcal{L} \mapsto \mathcal{M}_{\mathcal{L}}) \\
\big(\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\, \mathcal{M}^r_{\mathcal{L}}[a] > 0 \implies \mathcal{L}[a] = \mathcal{M}^r_{\mathcal{L}}[a] \times \texttt{sizeof}(r)\big) \wedge & (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{L}) \\
\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\, \mathcal{M}^r_{\mathcal{V}}[a] \implies & \\
\quad \big(\forall f \in \mathbb{F}(r).\, & \\
\qquad \mathcal{V}\big[a + \texttt{offsetof}(f)\big] \wedge & (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{V}) \\
\qquad \mathcal{M}^r_f[a] = \mathcal{B}\big[a + \texttt{offsetof}(f)\big]\big) & (\mathcal{M}_{\mathcal{B}} \leftrightarrows \mathcal{B}).
\end{array}
$$

Notation

- **Reference** evaluation state: $S_{\mathcal{R}}, \top$ or $\bot$, consistency invariants: $C_{\mathcal{R}}(S_{\mathcal{R}})$
- **Model** evaluation state: $S_{\mathcal{M}}, \top$ or $\bot$, consistency invariants: $C_{\mathcal{M}}(S_{\mathcal{R}})$
- **Correspondence invariants**: $S_{\mathcal{R}} \sim S_{\mathcal{M}}$

Core language

Syntax

$$
\begin{array}{lcl}
t_v & ::= & \\
 & | & i \\
 & | & p_1 \texttt{ - } p_2 \\
 & | & p_1 \texttt{ == } p_2 \\
 & | & p\texttt{->}f_v \\
t_p & ::= & \\
 & | & \texttt{NULL} \\
 & | & p \texttt{ + } v \\
 & | & p\texttt{->}f_p \\
 & | & \texttt{\&}p\texttt{->}f_s \\
 & | & \boldsymbol{container\_of}(p, f_s)
\end{array}
$$

$$
\begin{array}{lcl}
o & ::= & \\
 & | & v \texttt{ = } t_v \\
 & | & p \texttt{ = } t_p \\
 & | & p \texttt{ = } \boldsymbol{alloc}(n) \\
 & | & \boldsymbol{free}(p) \\
 & | & p\texttt{->}f_v \texttt{ = } v \\
 & | & p_1\texttt{->}f_p \texttt{ = } p_2 \\
 & | & \boldsymbol{assert}(c) \\
 & | & \boldsymbol{assume}(c) \\
 & | & \boldsymbol{havoc}(p, f_{pv}) \\
 & | & \boldsymbol{split}(p) \\
 & | & \boldsymbol{join}(p)
\end{array}
$$

$$
\begin{array}{lcl}
s & ::= & \\
 & | & \varepsilon \\
 & | & o; s
\end{array}
$$

Translation

- Introduce **empty structure** with tag `void_`
- Introduce **prefix field** `void_` of type `struct void_` to every structure, except `void_`

Translation

- Assume same transformations as in basic region-based model (eliminate & and * in favor of ->, wrap union fields in structures, if necessary)
- Translate non-prefix casts:

```
struct int_ *pn;
struct char_ *pc = (struct int_ *)pn;
                ⇓
struct char_ *pc =
  container_of(&pn->void_, struct char_, void_);
```

- Translate unions:

```
union { struct char_ c; struct int_ n; } *pu;
char_ pc = pu->c.charM;
                ⇓
struct void_ *pu;
struct char_ *pc = container_of(pu, struct char_, void_);
char c = pc->charM;
```

Soundness

Inference rules

- Reference semantics

$$\mathrm{OP}_{\mathcal{R}}\ \frac{P_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{R}}\ \blacktriangleright\ s|\ f_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})}$$

$$\mathrm{OP}^{\bot}_{\mathcal{R}}\ \frac{P^{\bot}_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{R}}\ \blacktriangleright\ \bot}$$

$$\mathrm{OP}^{\top}_{\mathcal{R}}\ \frac{P^{\top}_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{R}}\ \blacktriangleright\ \top}$$

- Model semantics

$$\mathrm{OP}_{\mathcal{M}}\ \frac{P_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{M}}\ \rhd\ s|\ f_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})}$$

$$\mathrm{OP}^{\bot}_{\mathcal{M}}\ \frac{P^{\bot}_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{M}}\ \rhd\ \bot}$$

$$\mathrm{OP}^{\top}_{\mathcal{M}}\ \frac{P^{\top}_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})}{\boldsymbol{op}(\overline{v});\ s|\ S_{\mathcal{M}}\ \rhd\ \top}$$

- Progress: $\forall S, \overline{v}.$
  either $\neg P(S,\overline{v}) = P^{\bot}(S,\overline{v})$ or $\neg P(S,\overline{v}) = P^{\top}(S,\overline{v})$

Proof

For every OP: $\forall S_{\mathcal{R}}, S_{\mathcal{M}}, \overline{v}.\ C_{\mathcal{R}}(S_{\mathcal{R}}) \wedge C_{\mathcal{M}}(S_{\mathcal{M}}) \wedge S_{\mathcal{R}} \sim S_{\mathcal{M}} \Rightarrow$

- $P^{\bot}_{\mathcal{R}}(S_{\mathcal{R}},\overline{v}) \implies P^{\bot}_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})$
- $P^{\top}_{\mathcal{M}}(S_{\mathcal{M}},\overline{v}) \implies P^{\top}_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})$
- $P_{\mathcal{M}}(S_{\mathcal{M}},\overline{v}) \implies P_{\mathcal{R}}(S_{\mathcal{R}},\overline{v}) \wedge$
  $C_{\mathcal{R}}\big(f_{\mathcal{R}}(S_{\mathcal{R}},\overline{v})\big) \wedge C_{\mathcal{M}}\big(f_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})\big) \wedge f_{\mathcal{R}}(S_{\mathcal{R}},\overline{v}) \sim f_{\mathcal{M}}(S_{\mathcal{M}},\overline{v})$

Soundness

Result

$\forall S_{\mathcal{R}}, S_{\mathcal{M}}, s.\ S_{\mathcal{R}} \sim S_{\mathcal{M}} \implies s|\ S_{\mathcal{R}} \blacktriangleright^* \bot \implies s|\ S_{\mathcal{M}} \rhd^* \bot$
(sound approximation)

- Model is **sound** for **any** region analysis
- No completeness, even for earlier supported programs

Completeness

Provide user annotations

- Spurious separation between regions of the same type
  - Enforced region unification (for pointer expressions $p_1, \ldots, p_n$)

    `//@ may_alias(`$p_1, \ldots, p_n$`);`

- Separation between regions of different structure types
  - Reinterpretation annotations

    `//@ split(`p`);` `//@ join(`p`);`

- Coarse granularity of model semantics

$$\forall a \in \mathbb{B}^d, r \in \mathbb{R}.\ \mathcal{M}^r_{\mathcal{V}}[a] \Longrightarrow \forall f \in \mathbb{F}(r).\ \mathcal{V}\left[a + \texttt{offsetof}(f)\right] \quad (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{V})$$

  - Detached fields (same transformation as elimination of &)

```
struct s {
  int a;
  int d //@ detached
};
```

$\Rightarrow$

```
struct s {
  int a;
  struct int_ {
    int intM;
  } d;
};
```

Completeness

Proof

It's possible to *simulate* reference semantics within model semantics:

- Core language allows at most **one** memory access per statement
- Unify all regions of the same type with `may_alias`
- Prefix every statement with the corresponding `join`
- Put the corresponding `split` after every statement
- Make all fields of simple (non-composite) types `detached`

Big worst-case annotation overhead

Practical annotation overhead

Custom security module ≈3.5 KLoC

Preliminary estimate

- 12 out of 255 functions need additional annotations
- **31** additional annotations
  - Memory reinterpretation between integral types of different size, byte reordering (6 functions $\times$ 4 annotations $=$ **24**)
  - Flexible array members (4 functions $\times$ 1 annotation $=$ **4**)
  - Memory reinterpretation between a structure type and an array of `unsigned char` (1 function $\times$ 2 annotations $=$ **2**)
  - Pointer type cast of the form $(t$ `**`$)p$ where p has type `void **` and t is not `void` (1 function $\times$ 1 annotation $=$ **1**)
- Issues marked with • can be solved automatically without annotations, reducing the number of annotations to **26**

Future work

New annotations: region ascription :> and move

- Call site region split

```
void swap(int *a, int *b) { ... }
// ...
//@ ghost int *t;
//@ move(a[1], t);
swap(a[0], a[1] /*@ :> t */);
```

- Redundant unification

```
int *p = &a;
handle(p);
p /*@ :> &b */ = &b;
handle(p /*@ :> &b */);
```

Can also be addressed by preliminary SSA transformation

Efficient translation of move is currently left for future work

Contents

Combined linear/bitwise reasoning

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x ^= y;
  y ^= x;
  x ^= y;
}
```

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x ^= y;
  y ^= x;
  x ^= y;
}
```

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

```
x + y <= 4 && x - y >= 2
```

Linear reasoning (just an illustration)

$$\begin{cases} x + y \leq 4, \\ x - y \geq 2 \end{cases} \iff \begin{cases} x + y \leq 4, \\ -x + y \leq -2 \end{cases} \iff \begin{cases} x + y \leq 4, \\ 2y \leq 2 \end{cases} \Rightarrow$$

$$y \leq 1$$

Combined linear/bitwise reasoning

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x ^= y;
  y ^= x;
  x ^= y;
}
```

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

```
x ^= y;
y ^= x;
x ^= y;
```

Bitvector reasoning (just an illustration)

$$\begin{cases} x' = (x \mathbin{\hat{}} y) \mathbin{\hat{}} \big(y \mathbin{\hat{}} (x \mathbin{\hat{}} y)\big), \\ y' = y \mathbin{\hat{}} (x \mathbin{\hat{}} y) \end{cases} \underset{\text{extensionality}}{\Longleftrightarrow} \begin{cases} \forall i \in [0{,}31] \,.\, x'_i = (x_i \oplus y_i) \oplus \big(y_i \oplus (x_i \oplus y_i)\big), \\ \forall i \in [0{,}31] \,.\, y'_i = y_i \oplus (x_i \oplus y_i) \end{cases} \Leftrightarrow$$

$$\begin{cases} \forall i \in [0,31] \,.\, x'_i = y_i, \\ \forall i \in [0,31] \,.\, y'_i = x_i \end{cases} \underset{\text{extensionality}}{\Longleftrightarrow} \begin{cases} x' = y, \\ y' = x \end{cases}$$

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

```
x + y <= 4 && x - y >= 2
x - 1 <= 0
```

```
x ^= y;
y ^= x;
x ^= y;
```

Combined reasoning (just an illustration)

$$\begin{cases} y \leq 1, \\ x' = y, \\ y' = x \end{cases} \implies x' \leq 1 \implies x' - 1 \leq 0$$

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

Mathematical encoding (ACSL semantics and WP)

$$
\begin{array}{l}
\forall x, y \in \mathbb{Z}. \\
\quad x + y \leq 4 \wedge x - y \geq 2 \wedge \mathrm{b}^{\mathtt{u}}_{32}(x) \wedge \mathrm{b}^{\mathtt{u}}_{32}(y) \implies \\
\quad \textbf{let } x_1 = \mathrm{c}^{\mathtt{u}}_{32}(x \mathbin{\hat{}} y) \textbf{ in} \\
\quad \textbf{let } y_1 = \mathrm{c}^{\mathtt{u}}_{32}(y \mathbin{\hat{}} x_1) \textbf{ in} \\
\quad \textbf{let } x_2 = \mathrm{c}^{\mathtt{u}}_{32}(x_1 \mathbin{\hat{}} y_1) \textbf{ in} \\
\qquad x_2 - 1 \leq 0,
\end{array}
$$

where

$\mathrm{c}^{\mathtt{u}}_{32}(x)$ — $\mathbb{Z} \to$ `unsigned` cast (<u>axiomatically defined</u> on $\mathbb{Z}$)

$\mathrm{b}^{\mathtt{u}}_{32}(x) \equiv 0 \leq x < \mathtt{UINT\_MAX}$

$x \mathbin{\hat{}} y$ — bitwise OR <u>axiomatically</u> defined on $\mathbb{Z}$ (as ininite sequences of bits)

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

Bitwise encoding (some software model checkers, e.g. CBMC, CPAchecker)

$$
\begin{array}{l}
\forall x, y \in \mathbb{B}^{32}. \\
\quad x \mathbin{\widehat{+}} y \mathbin{\widehat{\leq}^{\mathtt{u}}} 4_{32} \wedge x \mathbin{\widehat{-}} y \mathbin{\widehat{\geq}^{\mathtt{u}}} 2_{32} \implies \\
\quad \textbf{let } x_1 = x \mathbin{\widehat{\oplus}} y \textbf{ in} \\
\quad \textbf{let } y_1 = y \mathbin{\widehat{\oplus}} x_1 \textbf{ in} \\
\quad \textbf{let } x_2 = x_1 \mathbin{\widehat{\oplus}} y_1 \textbf{ in} \\
\qquad x_2 \mathbin{\widehat{-}} 1_{32} \mathbin{\widehat{\leq}^{\mathtt{u}}} 0_{32},
\end{array}
$$

where

$$
\begin{array}{lcl@{\qquad}lcl}
\mathbb{B}^{32} & = & \texttt{(\_ BitVec 8)} & n_{32} & \equiv & \texttt{(\_ bv}n\texttt{ 32)} \\
\widehat{+} & = & \texttt{bvadd} & \widehat{\leq}^{\mathtt{u}} & = & \texttt{bvule} \\
\widehat{-} & = & \texttt{bvsub} & \widehat{\geq}^{\mathtt{u}} & = & \texttt{bvuge} \\
\widehat{\oplus} & = & \texttt{bvxor} & & &
\end{array}
$$

Sample function

Combined linear/bitwise reasoning

Mathematical encoding (ACSL semantics and WP)

- Many quantifier instantiations needed for bitwise reasoning $\Rightarrow$
- Inefficient bitwise reasoning $\Longrightarrow$
- Cannot prove post-condition `x - 1 <= 0` automatically

Bitwise encoding (some software model checkers, e.g. CBMC, CPAchecker)

- Theory of bitvectors is inefficient for reasoning about linear inequalities $\Rightarrow$
- Inefficient linear reasoning $\Longrightarrow$
- Cannot prove post-condition `x - 1 <= 0` automatically

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

Combined encoding
(AstraVer, fork of Jessie, not Jessie3)

$$
\begin{array}{l}
\forall x, y \in \mathbb{B}^{32}. \\
\quad \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y) \leq 4 \wedge \mathrm{i}^{\mathtt{u}}_{32}(x) - \mathrm{i}^{\mathtt{u}}_{32}(y) \geq 2 \implies \\
\quad \textbf{let } x_1 = x \mathbin{\widehat{\oplus}} y \textbf{ in} \\
\quad \textbf{let } y_1 = y \mathbin{\widehat{\oplus}} x_1 \textbf{ in} \\
\quad \textbf{let } x_2 = x_1 \mathbin{\widehat{\oplus}} y_1 \textbf{ in} \\
\qquad \mathrm{i}^{\mathtt{u}}_{32}(x_2) - 1 \leq 0,
\end{array}
$$

where

$\mathrm{i}^{\mathtt{u}}_{32}(x) : \mathbb{B}^{32} \to \mathbb{Z}$ — uninterpreted function,
can use `bv2int`/`bv2nat`,
but they are poorly supported

Integer models: mathematical, bitwise and composite

Combined linear/bitwise reasoning

Combined encoding
(AstraVer, fork of Jessie, not Jessie3)

Consequences for SMT solvers:

- Theory of bitvectors propagates $x_2 = y$
- Theory of uninterpreted functions propagates $\mathrm{i}^{\mathtt{u}}_{32}(x_2) = \mathrm{i}^{\mathtt{u}}_{32}(y)$
- Theory of integers propagates $\mathrm{i}^{\mathtt{u}}_{32}(y) \leq 1$
- Can prove post-condition `x - 1 <= 0` ($\mathrm{i}^{\mathtt{u}}_{32}(x_2) - 1 \leq 0$) automatically

Contents

Operation semantics: checked unbounded vs. wrap-around

Combined reasoning for linear operations

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x += y;
  y = x - y;
  x -= y;
}
```

Operation semantics: checked unbounded vs. wrap-around

Semantics of linear operations

```
x += y;
```

- Checked unbounded semantics
  $\mathbf{let}\ x_1 = x + y\ \mathbf{in}\ \ldots \wedge$
  $0 \leq x + y \leq \texttt{UINT\_MAX}$
- Wrap-around semantics
  $\mathbf{let}\ x_1 = x \mathbin{\widehat{+}} y\ \mathbf{in}\ \ldots$

Operation semantics: checked unbounded vs. wrap-around

Semantics of linear operations

```
x += y;
```

Combined semantics

- No overflow, unbounded semantics
  $\mathbf{let}\ x_1 = \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y)\ \mathbf{in}\ \ldots \wedge$
  $0 \leq \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y) \leq \mathtt{UINT\_MAX}$
- Overflow, wrap-around semantics
  $\mathbf{let}\ x_1 = x \mathbin{\widehat{+}} y\ \mathbf{in}$
  $\mathrm{i}^{\mathtt{u}}_{32}(x_1) = \left(\mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y)\right) \bmod (\mathtt{UINT\_MAX} + 1) \wedge \ldots,$
  where mod is defined <u>axiomatically</u>

Operation semantics: checked unbounded vs. wrap-around

Semantics of linear operations

```
x += y;
```

Combined semantics, conflicting alternatives

- No overflow, unbounded semantics
  $\mathbf{let}\ x_1 = \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y)\ \mathbf{in}\ \ldots \wedge$
  $0 \leq \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y) \leq \mathtt{UINT\_MAX}$
  - Efficient linear reasoning
  - No bitwise reasoning and support for wrap-around semantics
- Overflow, wrap-around semantics
  $\mathbf{let}\ x_1 = x \mathbin{\widehat{+}} y\ \mathbf{in}$
  $\mathrm{i}^{\mathtt{u}}_{32}(x_1) = \left(\mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y)\right) \bmod (\mathtt{UINT\_MAX} + 1) \wedge \ldots,$
  where mod is defined axiomatically
  - Bitwise reasoning and support for wrap-around semantics
  - Poor linear reasoning

Operation semantics: checked unbounded vs. wrap-around

Semantics of linear operations

```
x += y;
```

Combined semantics. Solution

- Provide both semantics
- Introduce new operation

```
x +/*@%*/ y
(x +=/*@%*/ y;)
```

- Same solution for other operations

```
(+/*@%*/, * /*@%*/, / /*@%*/,
++/*@%*/, --/*@%*/, +=/*@%*/,
*=/*@%*/, ...)
```

Operation semantics: checked unbounded vs. wrap-around

Combined reasoning for linear operations

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures x - 1 <= 0;
  @*/
void f(void)
{
  x +=/*@%*/ y;
  y = x -/*@%*/ y;
  x -=/*@%*/ y;
}
```

Can be verified automatically even without theory of bit-vectors (e.g. by Alt-Ergo)

Contents

Real world examples: wrap-around arithmetic & casts (1)

Linux kernel (lib/string.c, memchr)

```
void *memchr(const void *s, int c, size_t n)
{
        const unsigned char *p = s;
        while (n-- != 0) {
                if ((unsigned char)c == *p++) {
                        return (void *)(p - 1);
                }
        }
        return NULL;
}
```

Real world examples: wrap-around arithmetic & casts (1)

Linux kernel (lib/string.c, memchr)

```
void *memchr(const void *s, int c, size_t n)
{
        const unsigned char *p = s;
        while (n--/*@%*/ != 0) {
                if ((unsigned char)/*@%*/c == *p++) {
                        return (void *)(p - 1);
                }
        }
        return NULL;
}
```

- Wrap-around integer arithmetic
- Intentionally overflowing cast

Real world examples: wrap-around arithmetic & casts (2)

Linux kernel (lib/string.c, strncasecmp)

```
int strncasecmp(const char *s1, const char *s2, size_t len)
{
        unsigned char c1, c2;
        if (!len) return 0;
        do {
                c1 = *s1++;
                c2 = *s2++;
                if (!c1 || !c2) break; if (c1 == c2) continue;
                c1 = tolower(c1); c2 = tolower(c2);
                if (c1 != c2) break;
        } while (--len);
        return (int)c1 - (int)c2;
}
```

Real world examples: wrap-around arithmetic & casts (2)

Linux kernel (lib/string.c, strncasecmp)

```
int strncasecmp(const char *s1, const char *s2, size_t len)
{
        unsigned char c1, c2;
        if (!len) return 0;
        do {
                c1 = /*@(unsigned char %)*/*s1++;
                c2 = /*@(unsigned char %)*/*s2++;
                if (!c1 || !c2) break; if (c1 == c2) continue;
                c1 = tolower(c1); c2 = tolower(c2);
                if (c1 != c2) break;
        } while (--len);
        return (int)c1 - (int)c2; // <-- Prevent overflow on '-'
}
```

- Implicit intentionally overflowing cast
- No single integer model (math, defensive or modulo), per-operation annotations needed

Contents

Composite integer model: summary

New annotations

- Arithmetic operations

```
+/*@%*/
```

- Compound assignment operators

```
+=/*@%*/
```

- Postfix operators

```
++/*@%*/
```

- Explicit casts

```
(unsigned char)/*@%*/
```

- Implicit casts (also checks the specified target type)

```
/*@(unsigned char %)*/
```

Comparison operations in code have double semantics

$$\mathtt{x} < \mathtt{y} \quad \rightarrow \quad \mathrm{i}^{\mathtt{u}}_{32}(x) < \mathrm{i}^{\mathtt{u}}_{32}(y) \wedge x \widehat{<}^{\mathtt{u}} y$$

Bounded integer operations in logic

What about logic?

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x +=/*@%*/ y;
  y = x -/*@%*/ y;
  x -=/*@%*/ y;
}
```

Can't prove the function directly, need auxiliary assertions $\Longrightarrow$ need bitwise operations in logic

Bounded integer operations in logic

Typing

```
unsigned x, y;
int z;
```

In ACSL: implicit casts to `integer` ($\mathbb{Z}$) for <u>every term</u>

$$\texttt{x} + \texttt{y} \to x + y, \quad \texttt{x} + \texttt{z} \to x + z,$$
$$\texttt{x} \,\hat{}\, \texttt{y} \to x \,\hat{}\, y, \quad \texttt{x} \,\hat{}\, \texttt{z} \to x \,\hat{}\, z,$$

$$x, y, z \in \mathbb{Z}$$

Bounded integer operations in logic

Typing

```
unsigned x, y; int z;
```

In modified ACSL ($x, y, z \in \mathbb{B}^{32}$):

- implicit cats to `integer` ($\mathbb{Z}$) on arithmetic operations

  `x + y` $\rightarrow \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(y)$, `x + z` $\rightarrow \mathrm{i}^{\mathtt{u}}_{32}(x) + \mathrm{i}^{\mathtt{u}}_{32}(z)$

- precise matching of types on bitwise and wrap-around operations

  `x ^ y` $\rightarrow x \widehat{\oplus} y$, `x +% y` $\rightarrow x \widehat{+} y$

  `x +% z` $\rightarrow$ typing error

- numeric constants are casted implicitly, if in the range

  `x +% 1` $\rightarrow x \widehat{+} 1_{32}$ `1 +% 1` $\rightarrow$ typing error

- comparison operations are "ad-hoc polymorphic", i.e. semantics depend on the type of the operands

  `x < 1` $\rightarrow x \widehat{<}^{\mathtt{u}} 1_{32}$, `x < (`$\mathbb{Z}$`)1` $\rightarrow \mathrm{i}^{\mathtt{u}}_{32}(x) < 1$

Bounded integer operations in logic

Auto-instantiated axiomatic definitions

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x += /*@%*/ y;
  y = x -/*@%*/ y;
  x -= /*@%*/ y;
  /*@ assert x <= (integer)1; */
  /*@ assert x <= 1; */
}
```

What about auto-instantiating axiom

`\forall unsigned x; x <= 1 <==> x <= (integer) 1;`

for every occurrence of =>?

Bounded integer operations in logic

Solution using ghost code

```
unsigned x, y;
/*@ requires x + y <= 4 && x - y >= 2;
  @ ensures (x & 2) == 0;
  @*/
void f(void)
{
  x += /*@%*/ y;
  y = x -/*@%*/ y;
  x -= /*@%*/ y;
  //@ ghost int b = x <= 1;
  /*@ assert b; */
}
```

Contents

Proofs as C code

Advantages

- Shallower learning curve
  no need to study complex reasoning tools e.g. Coq)
- Maintainability
  code and proofs in the same codebase
- Simpler proofs
  reuse of the translation engine (integer and memory models), more capable decision procedures

Disadvantages

- Bloating specification language (ACSL) with many features would make it more complicated
- Duplicated functionality
  Coq may be still needed for some complex proofs, e.g. existing
- Only suitable for deductive verification plugins
- Poorer reproducibility of proofs (on different ATP versions, different machines)

Proofs as C code

What's needed

- ☑ Annotations in ghost code (`/@ ... @/`)
- ☐ Allow logic types in ghost code
- ☐ Lemma functions with multiple labels
- ☐ Axiomatic inclusion/cloning with type and logic substitution
- ☐ Support auto-instantiation of statements (axioms, lemmas) triggered by logic symbol application

Proofs as C code

Results so far

- ☑ Tricky average computation

```
(a & b) + ((a ^ b) >> 1U)
```

- ☑ Kernighan's bitcount
- ☑ Gray codes (SWAR decoding and properties)
- ☐ n queens
  (from the paper *Verifying Two Lines of C with* WHY3)

Summary

- Linux kernel code make extensive use of generic pointer casts, pointer arithmetic, bitwise and wrap-around operations
- Generic memory handling can be supported by user-guided separation analysis without losing efficiency relative to the region-based model
- Support for composite integer model combined with ghost functions allows to efficiently reason about non-trivial code fragments involving bitwise operations without the use of external interactive tools (e.g. Coq or Isabelle)
- Further ACSL extensions are needed for more complex specifications (e.g. n queens) and better support for framing (moving pointer sets between regions)

ACSL modifications

- Reinterpretation annotations (`split`, `join`)
- Unification annotation (`may_alias`)
- Detached (addressable) fields (`detached`)
- More annotations for better framing (?)
- Wrap-around annotations for linear operations (`/*@%*/`) and implicit casts (`/*@(`t `%)*/`)
- Wrap-around operations in logic (`+%`, `-%`, `/%`, etc.)
- Nested annotations (`/@ ... @/`)
- Parametrized axiomatic inclusion
  (... `include` A { `type` t = u;
  `logic` t f ($\overline{x}$) = g ($\overline{x}$); ... })

Thank you!

Contacts

ISP RAS

Mikhail Mandrykin

email: mandrykin@ispras.ru

Alexey Khoroshilov

email: khoroshilov@ispras.ru