Slicing and Scope-Bounded Verification with Polymorphic Region and Effect Inference

Mikhail Mandrykin

ISP RAS

ISP RAS, September 26th, 2018

Contents

Issues

Target programs

Large programs, obtained from industrial software systems, $\gtrsim$ 10KLoc

Target issues

- Relatively large proportion of **irrelevant code**, typically simple abstractions suffice to efficiently prune the search space
- Analysis is typically **global** i.e. function calls are **inlined** at least once (for BAM)
- Incomplete code prompts issues with pointer analysis, especially **assignments through unknown pointers** that in the worst case can invalidate the entire abstract state

Overview

Suggested approach

Employ a **fast**, **local** and **sound** separation analysis, even if it is a relatively crude (imprecise) one

Solutions

- **Slicing** based on effect inference and interference
- **Scope-bounded verification** by skipping function calls soundly approximated with non-deterministic effects
- Avoid complete loss of precision by **strengthening** the global analyses with the results of the preliminary local separation analysis

Contents

Core language

t_n	$::=$		**numeric term**
	$\mid$	v	numeric variable
	$\mid$	$a(t_n, \ldots, t_n)$	arithmetic expression on numeric terms
	$\mid$	$*$	non-deterministic value
	$\mid$	t_{p1} - t_{p2}	pointer difference
	$\mid$	t_p->f_n	numeric field access

t_p	$::=$		**pointer term**
	$\mid$	p	pointer variable
	$\mid$	`NULL`	`NULL` pointer
	$\mid$	$\star$	start of a fresh memory block
	$\mid$	t_p + t_n	pointer shift
	$\mid$	t_p->f_p	pointer field access
	$\mid$	&t_p->f_s	nested structure
	$\mid$	$\boldsymbol{container\_of}(t_p, f_s)$	outer structure (field names are globally unique)

```
(struct outer *)((char *)inner
                 - offsetof(outer, inner))
```

i	$::=$		**instruction**
	$\mid$	$t_n = t_n$	numeric assignment
	$\mid$	$t_p = t_p$	pointer assignment
	$\mid$	$\boldsymbol{assert}\big(a(t_n, \ldots, t_n)\big)$	assertion
	$\mid$	$\boldsymbol{assume}\big(a(t_n, \ldots, t_n)\big)$	assumption
	$\mid$	$\boldsymbol{f}\big(t_p, \ldots, t_p, t_n, \ldots, t_n\big)$	function call

Reduction (operators)

- Eliminate occurrences of the **lone address operator** `&` (i.e. without the arrow `->`)

```
struct s { int f; } s;
int v;
int *p = &v, u = v, *w = &s.f;
```

$\Longrightarrow$

```
struct s { int f; } *s =
          alloca(sizeof(struct s));
int *v = alloca(sizeof(int));
int *p = v, u = *v, w = &s->f;
```

- Eliminate occurences of the **dereference operator** `*`

```
int *p;
int v = *p;
```

$\Longrightarrow$

```
struct int_ { int f };
struct int_ *p;
int v = p->f;
```

- Eliminate occurences of the **dot operator** `.`

```
struct s { struct int_ f; } s;
int v = s.f.f;
```

$\Longrightarrow$

```
struct s { struct int_ f; } *s =
          alloca(sizeof(strut s));
int v = (&s->f)->f;
```

Sample program (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 }
```

Sample program (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 }
```

Reduction (unoins and pointer type casts)

- Introduce **empty structure** with tag `void_`
- Introduce **prefix field** `void_` of type `struct void_` to every structure, except `void_`
- Wrap union fields into structures (as with `int` → `struct int`)
- Translate non-prefix casts:

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

- Translate unions:

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

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)

$$\begin{aligned}
&\forall r_1, r_2 \in \mathbb{R}. \\
&\quad r_1 \neq r_2 \implies \\
&\quad \forall p_1 \in r_1, p_2 \in r_2. \\
&\qquad \big((\texttt{char *})p_1\big) + \big(0 \ldots \texttt{sizeof}(p_1)\big) \cap \\
&\qquad\quad \big((\texttt{char *})p_2\big) + \big(0 \ldots \texttt{sizeof}(p_2)\big) = \emptyset
\end{aligned}$$

Polymorphic separation analysis

Regions (identifiers of the expression sets)

r	$::=$		**region**
	$\mid$	v	variable or fresh block – root region
	$\mid$	$f_p(r)$	dereference region
	$\mid$	$f_s(r)$	nested region
	$\mid$	$f_s^{-1}(r)$	outer region

Congruence closure of a relation $\sim$ (least fix point)

$$\textsc{Refl}\quad r \sim r$$

$$\textsc{Symm}\quad \frac{r_1 \sim r_2}{r_2 \sim r_1}$$

$$\textsc{Trans}\quad \frac{r_1 \sim r_2 \qquad r_2 \sim r_3}{r_1 \sim r_3}$$

$$\textsc{Cong}\quad \frac{r_1 \sim r_2}{f(r_1) \sim f(r_2)}\ f \in \{f_p, f_s, f_s^{-1}\}$$

$$\textsc{Inv}\quad \frac{f_s(r) \sim r_1}{f_s^{-1}(r_1) \sim r}$$

Roots and kinds

$$root(v) = v$$

$$root\big(f(r)\big) = root(r), f \in \{f_p, f_s, f_s^{-1}\}$$

$$kind(r) = \begin{cases} \texttt{GLOBAL}, & \exists r' \,.\, r \sim r' \wedge root(r') \text{ is a global variable} \\ \texttt{POLY}, & \text{otherwise, if } \exists r' \,.\, r \sim r' \wedge root(r') \text{ is a parameter} \\ \texttt{LOCAL}, & \text{otherwise} \end{cases}$$

Polymorphic separation analysis

Substitutions

A function call $f(t_{p1}, \ldots, t_{pk}, t_{n1}, \ldots, t_{nl})$, where f has parameters $p_1, \ldots, p_k, p_{k+1}, \ldots p_{k+l}$ determines a substitution σ_f, such that $\sigma_f(p_1) = t_{p1}, \ldots, \sigma_f(p_k) = t_{pk}, \sigma_f(p_{k+1}) = t_{n1}, \sigma_f(p_{k+l}) = t_{nl}$.

Substitutions on regions

$$\begin{aligned} \sigma(v) &= \begin{cases} \sigma(p_i), & v \text{ is a parameter } p_i \\ v, & v \text{ is a global variable} \end{cases} \\ \sigma\big(f(r)\big) &= f\big(\sigma(r)\big), f \in \{f_p, f_s, f_s^{-1}\} \end{aligned}$$

Projection of a congruence relation

$\sim' = \{(r_1, r_2) | r_1 \sim r_2, kind(r_i) \in \{\texttt{GLOBAL}, \texttt{POLY}\}, i = 1, 2\}$

Substitutions on congruence relations

$$\sigma(\sim_f) = \left\{ \big(\sigma(r_1), \sigma(r_2)\big) \middle| r_1 \sim'_f r_2 \right\}$$

Polymorphic separation analysis

Regions of pointer terms

$$
\begin{aligned}
r(p) &= p \\
r(\texttt{NULL}) = r(\star) &= v^*, \text{where } v^* \text{ is a fresh dummy variable name} \\
r(t_p \texttt{ + } t_n) &= r(t_p) \\
r(t_p\texttt{->}f_p) &= f_p\big(r(t_p)\big) \\
r(\texttt{\&}t_p\texttt{->}f_s) &= f_s\big(r(t_p)\big) \\
r\big(\textbf{\textit{container\_of}}(t_p, f_s)\big) &= f_s^{-1}\big(r(t_p)\big)
\end{aligned}
$$

Separation analysis on instructions

$$
\begin{aligned}
&t_{n1} \texttt{ = } t_{n2} && — \\
&t_{p1} \texttt{ = } t_{p2} && \sim \leftarrow CC\Big(\sim \cup \big\{ \big(r(t_{p1}), r(t_{p2})\big) \big\} \Big) \\
&\textbf{\textit{assert}}\big(a(t_n, \ldots, t_n)\big) && — \\
&\textbf{\textit{assume}}\big(a(t_n, \ldots, t_n)\big) && — \\
&\boldsymbol{f}\big(t_p, \ldots, t_p, t_n, \ldots, t_n\big) && \sim \leftarrow CC\Big(\sim \cup \sigma_{\boldsymbol{f}}(\sim_{\boldsymbol{f}}) \Big),
\end{aligned}
$$

where $CC(R)$ is congruence closure of the relation R

Modularity of polymorphic separation analysis

Flow-sensitive analyses

```
// a ↪ a, b ↪ b, c ↪ c,
// a->f ↪ d, c->f ↪ e
a->f = b;
// a ↪ a, b ↪ b, c ↪ c,
// a->f ↪ b, c->f ↪ e
```

```
// a ↪ a, b ↪ b, c ↪ a,
// a->f ↪ d, c->f ↪ d
a->f = b;
// a ↪ a, b ↪ b, c ↪ a,
// a->f ↪ b, c->f ↪ b
```

- How to **quickly** compute results for $a \sim c$ based on the results for $a \nsim c$?
- No general solution for control-flow sensitive analyses . . .

Flow-insensitive analysis

```
// f(a) = b
a->f = b;
// f(a) = b
```

```
// f(a) = b ∧ a = c
a->f = b;
// f(a) = b ∧ a = c
```

- The summary $\sim$ is a conjunction of equality of constraints **closed under congruence**
- $f(a) = b \wedge a = c \implies f(c) = b$ by congruence closure
- There are **fast** and **incremental** congruence closure algorithms, e.g., [R. Nieuwenhuis and A. Oliveras "Proof-producing congruence closure", 2005], so just combine the constraints using the algorithm!

Polymorphic separation analysis

Reduced sample program

```
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 }
```

Polymorphic separation analysis

Sample program. Result of separation analysis

$x = \cdot, sM(x) = \cdot, y = \cdot, p(sM(x)) = \cdot;\ z = \cdot, px = \cdot, \ldots, p(z) = \cdot$

```
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 }
```

Dependence set inference

Control flow graph

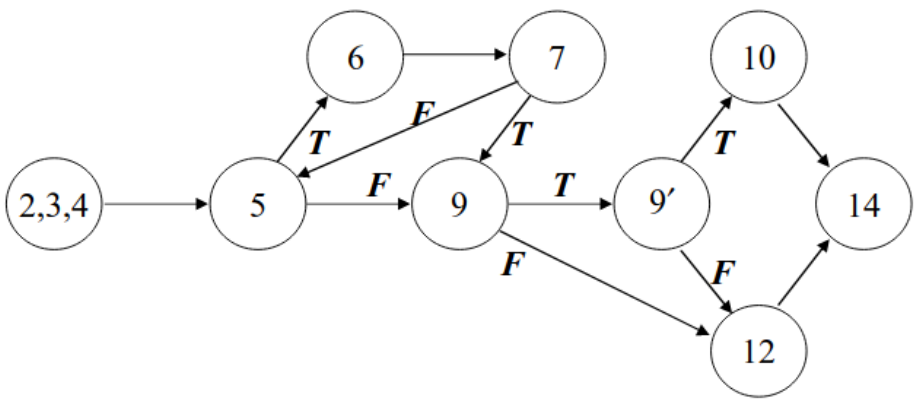
Consider control flow graph, where **statements** are represented by **vertices** and **states** (control flow transitions) are represented by **edges**. This is different from what is used in CPAchecker, but more convenient for the following definition

Dependence

Consider an `if` statement s with condition $a(t_{n1}, \ldots, t_{nk})$. Assume its **post-dominator** closest to s is the statement s_d. Then any statement s' different from both s and s_d on any path starting from s and finishing in s_d is called **dependent from** the condition $a(t_{n1}, \ldots, t_{nk})$. We write $a(t_{n1}, \ldots, t_{nk}) \in d(s')$

Dependence sets

Thus every statement s can be ascribed the corresponding dependence set $d(s)$ of conditions $a(t_{n1}, \ldots, t_{nk})$



For $s = 5$, $s_d = 9$.
So $a_5 \in d(6)$, $a_5 \in d(7)$.
For $s = 9$, $s_d = 14$.
So $a_9 \in d(9')$, $a_9 \in d(10)$, $a_9 \in d(12)$.
E.g. $d(10) = \{a_9, a_{9'}\}$

Effect inference

Access effects $e(t)$

Access effect is a set of variables (v or p),
fields acceses (pairs of the form (r, f_n) or (r, f_p)) and a special non-deterministic effect $*$

$$
\begin{aligned}
e(v) &= \{v\} \\
e\big(a(t_{n1}, \ldots, t_{nk})\big) &= \textstyle\bigcup_{i=1}^{k} e(t_{ni}) \\
e(*) &= \{*\} \\
e(t_{p1}\texttt{ - }t_{p2}) &= e(t_{p1}) \cup e(t_{p2}) \\
e(t_p\texttt{->}f_n) &= e(t_p) \cup \big(r(t_p), f_n\big)
\end{aligned}
\qquad
\begin{aligned}
e(p) &= \{p\} \\
e(\texttt{NULL}) &= \emptyset \\
e(\star) &= \{*\} \\
e(t_p\texttt{ + }t_n) &= e(t_p) \cup e(t_n) \\
e(t_p\texttt{->}f_p) &= e(t_p) \cup \big(r(t_p), f_p\big) \\
e(\texttt{\&}t_p\texttt{->}f_s) &= e(t_p)
\end{aligned}
$$

$$
e\big(\boldsymbol{container\_of}(t_p, f_s)\big) = e(t_p)
$$

Assignment effects $A(i)$

Assignment effects are sets of ordered pairs, where elements of pairs are access effects

$$
A(\underbrace{t_1\texttt{ = }t_2}_{i}) = \big\{(e_1, e_2) \mid e_1 \in e(t_1), e_2 \in e(t_2) \cup e\big(d(i)\big)\big\}
$$

$$
A\Big(\boldsymbol{assert}\big(a(t_{n1}, \ldots, t_{nk})\big)\Big) = \emptyset
$$

$$
A\Big(\underbrace{\boldsymbol{assume}\big(a(t_{n1}, \ldots, t_{nk})\big)}_{i}\Big) = \left\{(e_1, e_2) \;\middle|\; \begin{array}{l} e_1 \in e(t_{ni}), e_2 \in e(t_{nj}) \cup e\big(d(i)\big), \\ i, j \in \{1, \ldots, k\} \end{array} \right\}
$$

$$
A\Big(\underbrace{\boldsymbol{f}\big(t_p, \ldots, t_p, t_n, \ldots, t_n\big)}_{i}\Big) = \left\{(e_1, e_2) \;\middle|\; \begin{array}{l} \exists e_2' \cdot (e_1, e_2') \in \sigma_{\boldsymbol{f}}(A^*_{\boldsymbol{f}}), \\ (e_1, e_2) \in \sigma_{\boldsymbol{f}}(A^*_{\boldsymbol{f}}) \vee e_2 \in e\big(d(i)\big) \end{array} \right\}
$$

Effect inference

Access effects $e(t)$

Access effect is a set of variables (v or p),
fields acceses (pairs of the form (r, f_n) or (r, f_p)) and a special non-deterministic effect $*$

$$
\begin{aligned}
e(v) &= \{v\} \\
e\big(a(t_{n1}, \ldots, t_{nk})\big) &= \textstyle\bigcup_{i=1}^{k} e(t_{ni}) \\
e(*) &= \{*\} \\
e(t_{p1}\ \texttt{-}\ t_{p2}) &= e(t_{p1}) \cup e(t_{p2}) \\
e(t_p\texttt{->}f_n) &= e(t_p) \cup \big(r(t_p), f_n\big)
\end{aligned}
\qquad
\begin{aligned}
e(p) &= \{p\} \\
e(\texttt{NULL}) &= \emptyset \\
e(\star) &= \{*\} \\
e(t_p\ \texttt{+}\ t_n) &= e(t_p) \cup e(t_n) \\
e(t_p\texttt{->}f_p) &= e(t_p) \cup \big(r(t_p), f_p\big) \\
e(\texttt{\&}t_p\texttt{->}f_s) &= e(t_p)
\end{aligned}
$$

$$
e\big(\boldsymbol{container\_of}(t_p, f_s)\big) = e(t_p)
$$

Dependency effects $D(i)$

$$
\begin{aligned}
D(t_1 \texttt{ = } t_2) &= \emptyset \\
D\Big(\underbrace{\boldsymbol{assert}\big(a(t_{n1}, \ldots, t_{nk})\big)}_{i}\Big) &= e\big(a(t_{n1}, \ldots, t_{nk})\big) \cup e\big(d(i)\big) \\
D\Big(\boldsymbol{assume}\big(a(t_{n1}, \ldots, t_{nk})\big)\Big) &= \emptyset \\
D\Big(\underbrace{\boldsymbol{f}\,(t_p, \ldots, t_p, t_n, \ldots, t_n)}_{i}\Big) &= \sigma_{\boldsymbol{f}}(D^*_{\boldsymbol{f}}) \cup e\big(d(i)\big)
\end{aligned}
$$

Contents

Slicing

$A = \bigcup_i A(i), \quad D = \bigcup_i D(i)$

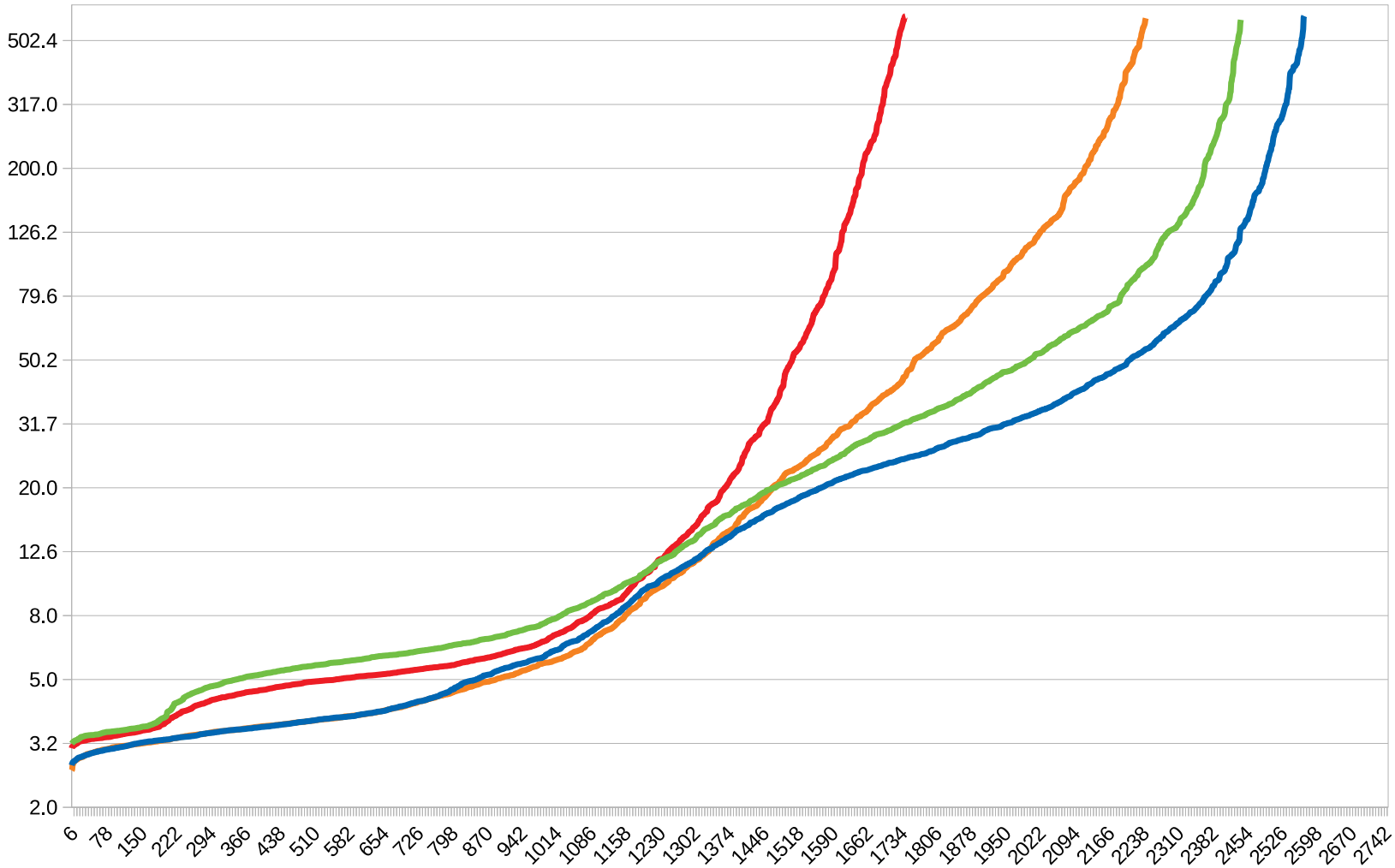
A^* is the transitive closure of A,
D^* is the closure of D under A^*

Slicing criterion

The instruction is **retained** iff $\mathrm{dom}\, A(i) \cap D^* \neq \emptyset$

Results

`SoftwareSystems` from SV-COMP'18, CPAchecker LDV and LDV-BAM, 400s for slicing, 600s in total, 8GB RAM



Strengthening other analyses

Other analyses e.g. predicate and explicit-value can be **strengthened** with regoin separation to improve precision in presence of **unknown pointers**. When assignment through an unknown pointer is encountered, the destination memory area can be **over-approximated by region**

Scope-bounded verification

- Over-approximate deeper functions with their stubs — non-deterministic **effects** (A^* and D^*) computed by the separation analysis
- Inline function bodies when a **counterexample** containing stubs is discovered
- Preliminary results show that additional summaries (e.g. symbolic execution) are needed to achieve significant sdpeedups
- There are ideas on function call **releancy detection** based on effects and path formulas

Thank you!

Contacts

ISP RAS

Mikhail Mandrykin

email: mandrykin@ispras.ru