Master’s Thesis

Solver-based Analysis of Memory Safety using Separation Logic

Moritz Beck

Software and Computational Systems Lab

LMU Munich

16.09.2020

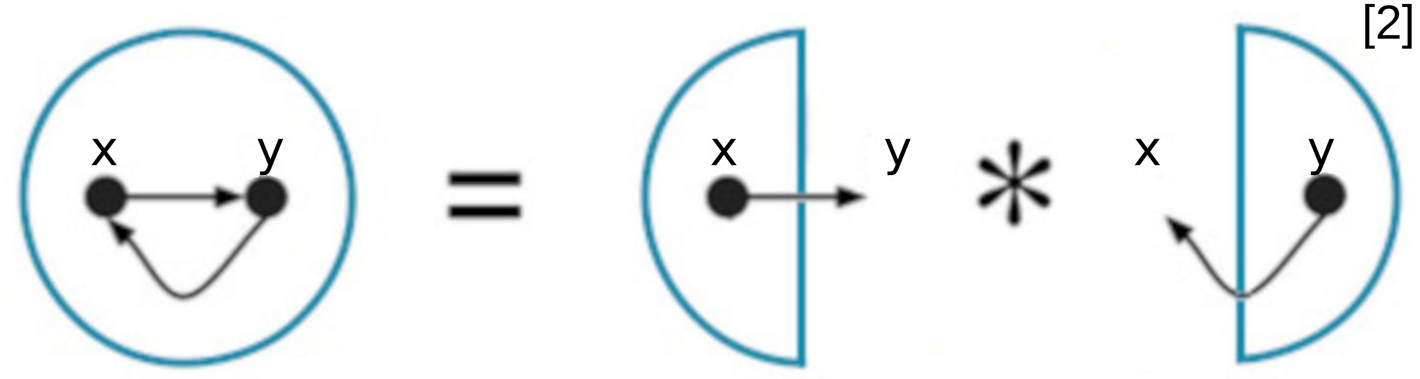
Motivation

”First, [SL] *merges with the scientific-engineering model the programmer uses to understand and build the software.* [...]

Secondly, the proof theory developed to check software using SL is based on rules for scaling the reasoning task [...]*”* (Pym et al. [1])

Background

- Two memory cells “separately in memory”



- Corresponding formula: $x \mapsto y \quad * \quad y \mapsto x$

- Implicit assertion by spatial conjuntion * that x ≠ y

Background

- Based on *Symbolic Heaps,* SL extends Hoare Logic [3, 4] by spatiality

- Rule of constancy does not hold for SL

$$\frac{\{x \mapsto 0\}\ \ {*x} = 42\ \{x \mapsto 42\}}{\{x \mapsto 0 \land y \mapsto 0\}\ \ {*x} = 42\ \{x \mapsto 42 \land y \mapsto \underset{42}{\cancel{0}}\}}$$

x=y

- Spatial conjunction works out

$$\frac{\{x \mapsto 0\}\ \ {*x} = 42\ \{x \mapsto 42\}}{\{x \mapsto 0 * y \mapsto 0\}\ \ {*x} = 42\ \{x \mapsto 42 * y \mapsto 0\}}$$

Pointer Analysis

- Based on symbolic execution
- CPA implementation of the CPAchecker framework
- Transformation of C code to SL formulae using JavaSMT
- Check formulae for memory safety properties:
  - Invalid read and write ($\perp^R, \perp^W$)
  - Invalid free ($\perp^F$)
  - Memory leak ($\perp^L$)

Pointer Analysis – Abstract Domain

$$
\begin{array}{rll}
\kappa & := \text{typical constants} & \text{Constants} \\
Var & := x, y, \ldots & \text{Program variables} \\
Var' & := x', y', \ldots & \text{Symbolic variables} \\
E, F, G & := \kappa \mid Var \mid Var' \mid \overbrace{E \odot F}^{\{+,\ -,\ <<,\ >>,\ \bullet\}} & \text{Expressions} \\
C & := true \mid E = F \mid E < F \mid \neg C & \text{Constraints} \\
\Pi & := C \mid \Pi \wedge \Pi & \text{Pure formulae} \\
\Sigma & := emp \mid E \mapsto F \mid \Sigma * \Sigma & \text{Spatial formulae} \\
\mathcal{H} & := \Pi \wedge \Sigma & \text{(Quantifier-free) Symbolic heaps}
\end{array}
$$

Grammar of SL formulae

Pointer Analysis – Language

$$
\begin{array}{rll}
\tau := & \text{typical types excluding floats} & \text{Types} \\
\kappa := & \text{typical constants} & \text{Constants} \\
\epsilon_\ell := & x, y, ... \mid \; * \epsilon \mid a[\epsilon] \mid a.b & \text{Left-hand sides} \\
\epsilon := & \kappa \mid \epsilon_\ell \mid \& \epsilon_\ell \mid \epsilon \odot \epsilon \mid f(\epsilon_r, ..., \epsilon_r) \mid (\tau)\epsilon & \text{Right-hand sides} \\
s := & \tau \;\; a \mid \epsilon \mid \epsilon_\ell = \epsilon \mid \{s\} \mid s; s & \text{Statements} \\
 & \mid \; while(\epsilon) \;\; s \mid if(\epsilon) \;\; s \;\; else \;\; s &
\end{array}
$$

Basic programming language inspired by C

Pointer Analysis – Locations and Values

- C assignment statement of the form $\epsilon_\ell = \epsilon$

$$\underbrace{\texttt{x}}_{\text{Location}} \texttt{ = } \underbrace{\texttt{x + 1}}_{\text{Value}}\texttt{;}$$

- Transformation of both to the representative formula

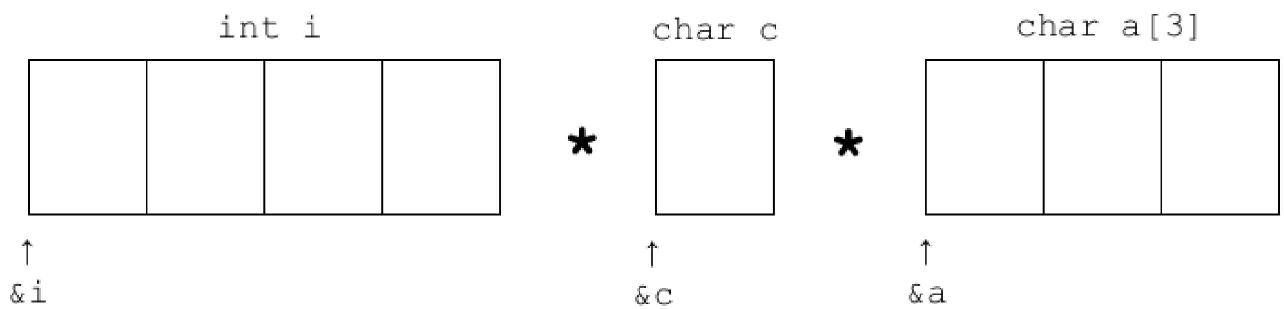
$$\Sigma \models \epsilon_\ell \Downarrow_l E \qquad \Sigma \models \epsilon \Downarrow_v E$$

Pointer Analysis – Memory Model

- Syntactic sugar for memory segments

$$E \mapsto F_0, ..., F_n := (E \mapsto F_0) * ... * (E + n \mapsto F_n)$$

- Memory as collection of byte-sequences



- Corresponding formula (simplified)

$$\&i \mapsto \underbrace{i^0, i^1, i^2, i^3}_{\text{bytes (little endian)}} \; * \; \&c \mapsto c \; * \; \&a \mapsto a[0], a[1], a[2]$$

Pointer Analysis – Memory Access

- Allocation check

$$\Sigma \models Allocated(E)$$
$$\Longleftrightarrow$$
$$\Sigma \models \exists \Sigma', F.(E = F \wedge \Sigma = \boxed{\Sigma'} * \boxed{F \mapsto -})$$

  - *Frame* and *antiframe*
  - Frame inference and antiframe abduction together refered to as *Bi-Abduction* [5]
  - Bi-Abduction <u>not</u> represented in SL-COMP 2019 [6]

Pointer Analysis – Memory Access

- Allocation check using a SL solver (without Bi-Abduction)

$$\neg SAT(\Sigma * E \mapsto -)$$

- Dereferencing

$$\begin{aligned}
Deref(emp,\ E) &:= nil \\
Deref(\Sigma, nil) &:= nil \\
Deref(F \mapsto G * \Sigma,\ E) &:= \text{if } \overbrace{F \mapsto G \models Allocated(E)}^{\neg SAT(F \mapsto G * E \mapsto -)} \\
&\quad\ \text{then } G \text{ else } Deref(\Sigma, E)
\end{aligned}$$

Pointer Analysis – Transfer Relation

- As part of the CPA formalism, CFA provides different kinds of edges:
  - Statement edge
  - Assumption edge
  - Function call edge
  - ...
- For each of them, rules are defined to track the memory manipulation

Pointer Analysis – Transfer Relation

- Statement edge: dynamic allocation and assignment

$$\frac{\begin{array}{c}\Sigma * E \mapsto \_^{0}, \ldots, \_^{n-1} \models \epsilon_\ell \Downarrow_l E \qquad n = sizeof(\epsilon_\ell) \\ \Sigma * E \mapsto \_^{0}, \ldots, \_^{n-1} \models \epsilon \Downarrow_v \kappa \qquad \Sigma \models \texttt{malloc}(\epsilon) \Downarrow_v x\end{array}}{\{\Pi \wedge \Sigma * \underbrace{E \mapsto \_^{0}, \ldots, \_^{n-1}}_{\textit{“E pointing to n bytes of an arbitrary value”}}\}\ \epsilon_\ell = \texttt{malloc}(\epsilon)\ \left\{\begin{array}{c}\Pi \wedge \Sigma * E \mapsto x^{0}, \ldots, x^{n-1} \\ * \, x \mapsto \_^{0}, \ldots, \_^{\kappa-1}\end{array}\right\}}\ \textsc{Malloc}$$

$$\frac{\Sigma \models \epsilon \Downarrow_v nil}{\{\Pi \wedge \Sigma\}\ \epsilon_\ell = \epsilon\ \{\bot^{R}\}}\ \textsc{Assign}^{\text{InvR}} \qquad \frac{\Sigma \models \epsilon_\ell \Downarrow_l nil}{\{\Pi \wedge \Sigma\}\ \epsilon_\ell = \epsilon\ \{\bot^{W}\}}\ \textsc{Assign}^{\text{InvW}}$$

Similar rule for statements
other than assignments

Pointer Analysis – Transfer Relation

- Statement edge: dynamic deallocation

$$\frac{\Sigma * E \mapsto -^0, \ldots, -^{n-1} \models \epsilon \Downarrow_v E \quad n = \mathit{segmentSize}(E)}{\{\Pi \wedge \Sigma * E \mapsto -^0, \ldots, -^{n-1}\}\ \texttt{free}(\epsilon)\ \{\Pi \wedge \Sigma\}}\ \textsc{Free}$$

$$\frac{\Sigma \models \epsilon \Downarrow_v E \quad \Sigma^h \not\models \mathit{Allocated}(E)}{\{\Pi \wedge \Sigma^s * \Sigma^h\}\ \texttt{free}(\epsilon)\ \{\bot^F\}}\ \textsc{Free}^{\mathrm{Inv1}}$$

$$\frac{\Sigma \models \epsilon \Downarrow_v E \quad \mathit{segmentSize}(E) = -1}{\{\Pi \wedge \Sigma\}\ \texttt{free}(\epsilon)\ \{\bot^F\}}\ \textsc{Free}^{\mathrm{Inv2}}$$

  - `segmentSize(E)` determines the size of an allocated segment for a given start address, `-1` otherwise
  - Symbolic heap can be subdivided into heap Σ^h and stack Σ^s part as referred to in the context of C

Pointer Analysis – Transfer Relation

- Assumption edge: feasability check

$$\frac{\Sigma \models \epsilon \Downarrow_v C \qquad \boxed{\Pi \wedge C \text{ is SAT}}}{\{\Pi \wedge \Sigma\}\ \epsilon\ \{\Pi \wedge C \wedge \Sigma\}}\ \textsc{Assume}^{+}$$

  - Ensures termination of analysis
  - SMT solver sufficient for satisfiability check

Pointer Analysis – Memory Leak

- Dropped values might lead to leakage (assignment, scope, `free()`)

```
char *p = 0;
{
  char *q = malloc(1);
  p = q;
}
```

Σ^s	Σ^h
p → 0 malloc!	malloc! → 0
q → malloc!	

Pointer Analysis – Memory Leak

- Dropped values pointing to an allocated heap cell have to be checked for reachability

$$Leak(\Sigma^s * \Sigma^h, E) := \overbrace{(\Sigma^h \models Allocated(E))}^{\text{E is active heap address}} \wedge \neg reachable(\Sigma^s * \Sigma^h, E)$$

$$\begin{aligned} reachable(\Sigma^s * \Sigma^h, E) := \; & \overbrace{(\Sigma^s \models Allocated(E))}^{\text{direct access via stack}} \\ & \vee (\underbrace{\exists F, G.(F \mapsto G \wedge E = G}_{\text{alias F exists}} \wedge \underbrace{reachable(\Sigma^s * \Sigma^h, F)}_{\text{check alias recursively}})) \end{aligned}$$

  - Remember already visited aliases to handle cycles (here: left out for readability)

Implementation

- Allocation checks for each heaplet might lead to overhead

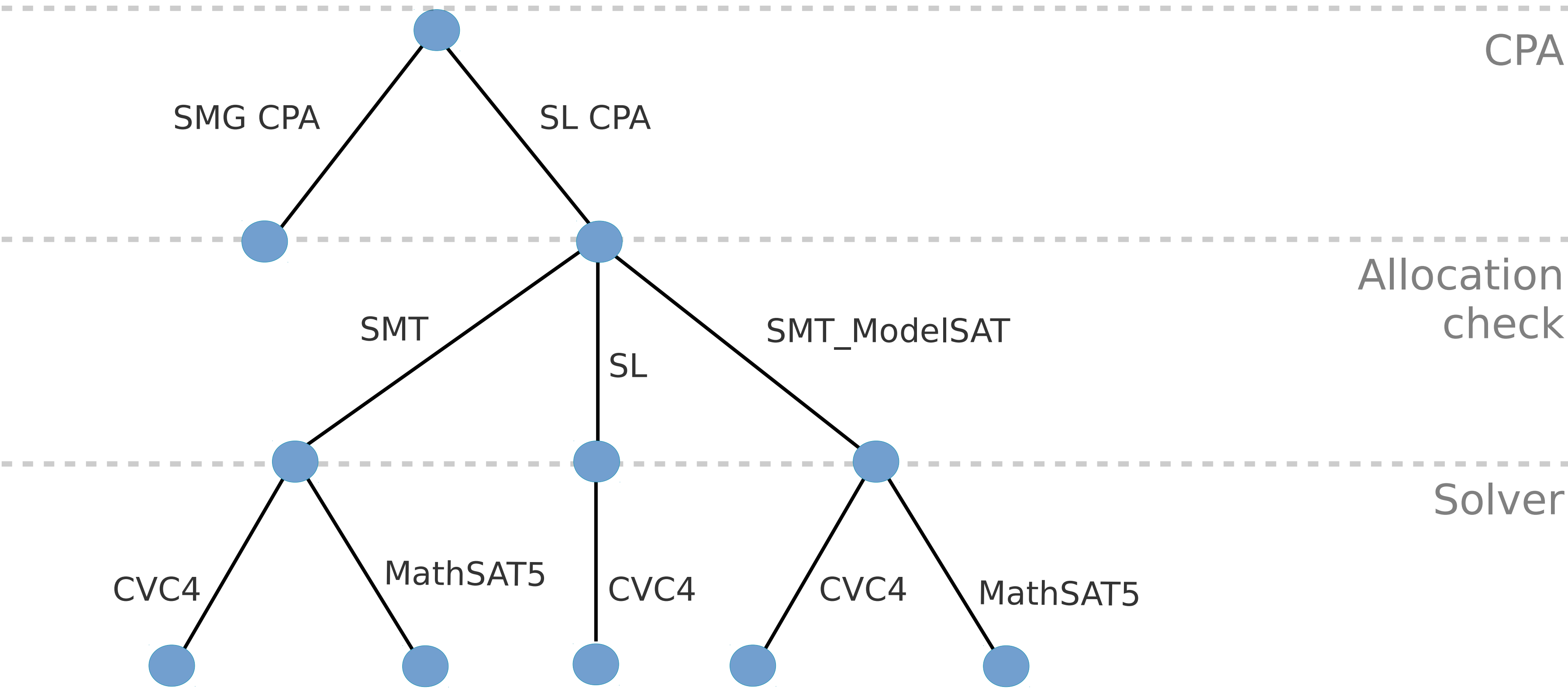
- “Simulation” through SMT

$$\neg SAT(F \mapsto G * E \mapsto -) \iff SAT(F = E)$$

- Can be further optimized to a single solver call using model generation

Evaluation

- Comparison of the approach (SL CPA) to SMG CPA (cf. [7])



Evaluation – Test-sets

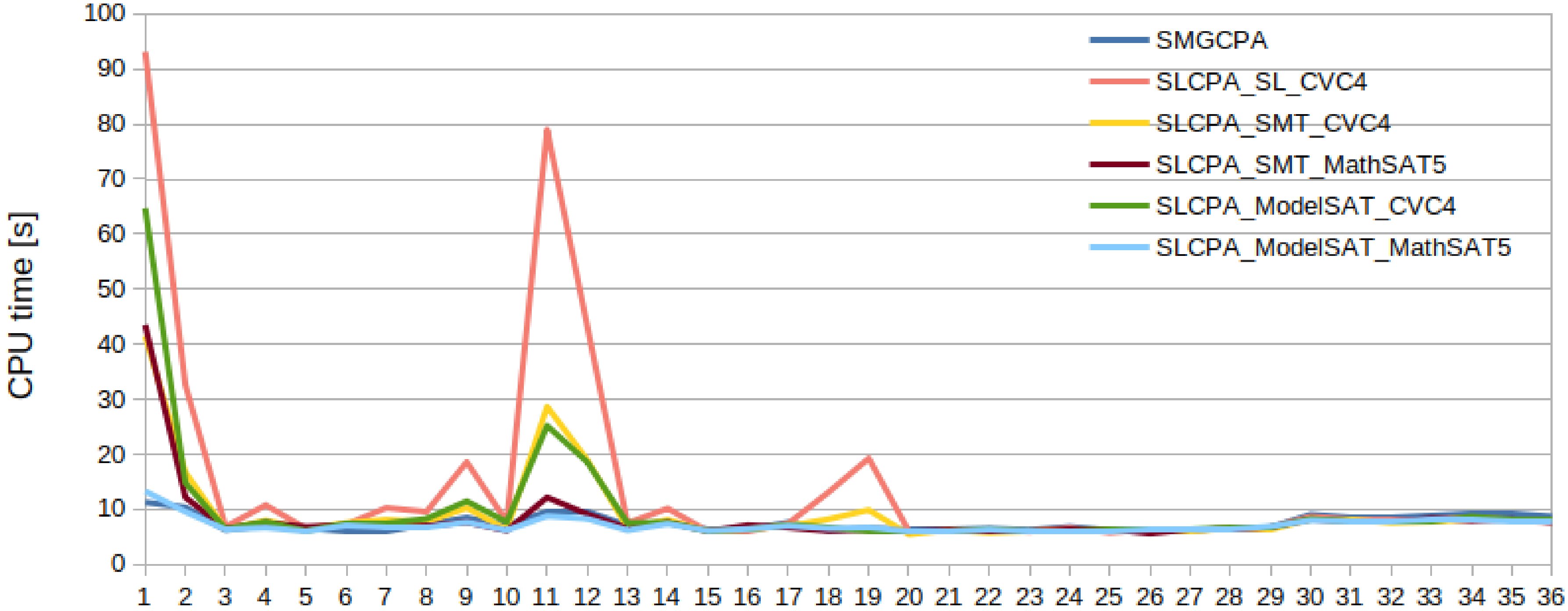
- SV-Benchmarks
  - memsafety-ext3 (18/18)[1]
  - memsafety-ext2 (2/10)[2]
- CPAlien test-set (16/21)[3]
- 36 problems solved
- including function calls, singly-linked lists and structures

1: https://github.com/sosy-lab/sv-benchmarks/tree/master/c/memsafety-ext3
2: https://github.com/sosy-lab/sv-benchmarks/tree/master/c/memsafety-ext2
3: https://github.com/sosy-lab/cpachecker/tree/trunk/test/programs/cpalien

Evaluation – Execution Environment

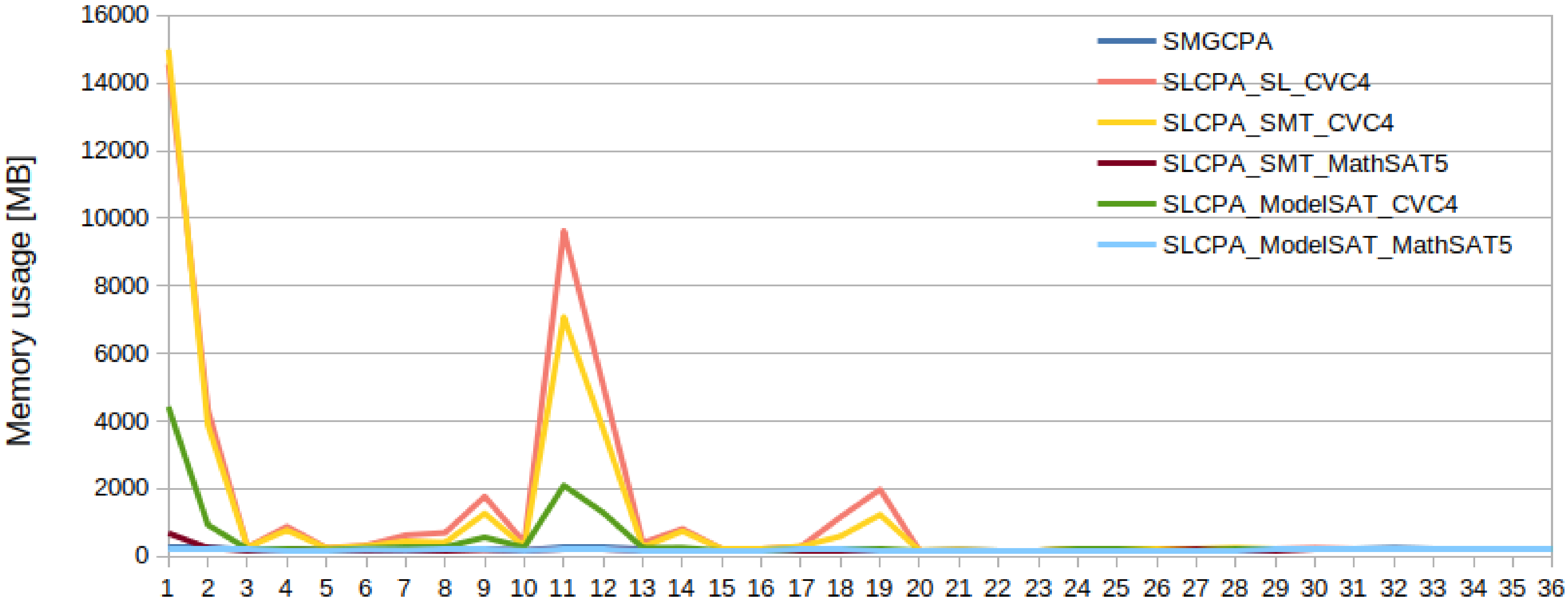
- Intel Xeon E3-1230 v5 CPUs, 3.40 GHz CPU frequency, 33 GB RAM
- Run on two CPU cores, limited to 90s execution time and 15GB RAM
- Measurments using BenchExec
- Branch: `sl-integration0:r34981`

Evaluation – CPU time



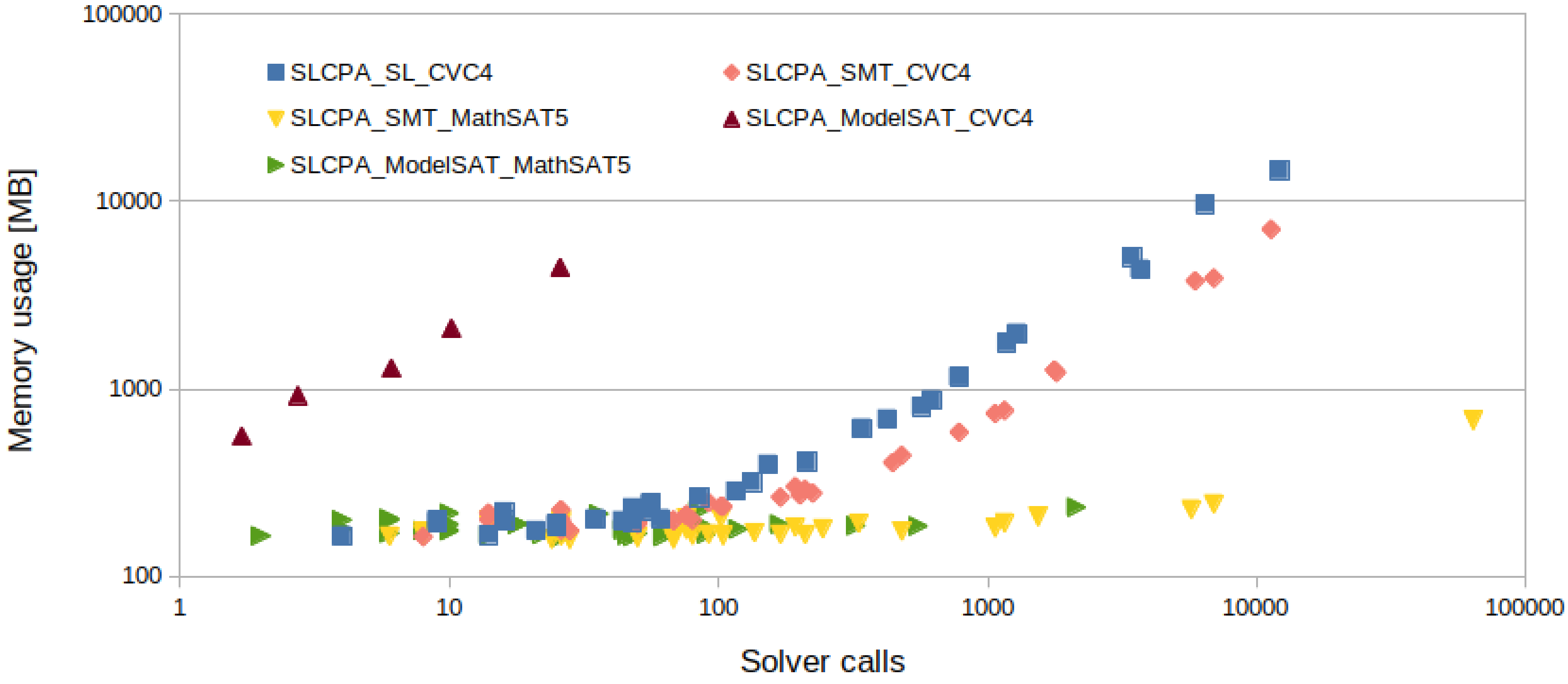
- SLCPA_SL_CVC4 slow and has one timeout
- All SMT aproaches significantly faster
- SLCPA_ModelSAT_MathSAT5 comparable to SMGCPA

Evaluation – Memory



- CVC4 without ModelSAT is memory consuming
- Again SLCPA_ModelSAT_MathSAT5 comparable to SMGCPA

Evaluation – Solver calls and Memory



- CVC4: significant increase of memory consumption with the amount of solver calls; not observable for MathSAT5

→ potential memory leak in CVC4 solver interface

Conclusion

- SL solver interface for allocation check and dereferencing is crucial for performance
- However, model checking approach with symbolic execution and SL worked out
- Problems without spatiality are better suited for SMT
- Combination of SL and SMT is promising in respect to composition and efficiency

References

[1] D. Pym, J. Spring, and P. O'Hearn. Why Separation Logic Works. Philosophy & Technology, 32(3):483–516, 2019. https://doi.org/10.1007/s13347-018-0312-8.

[2] Facebook Infer. https://fbinfer.com/docs/separation-logic-and-bi-abduction. Online; accessed 16.09.2020

[3] P. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In Computer Science Logic, pages 1–19. Springer, Berlin, Heidelberg, 2001. ISBN 978-3-540-44802-0. https://doi.org/10.1007/3-540-44802-0_1.

[4] D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In Proc. TACAS, pages 287–302. Springer, Berlin, Heidelberg, 2006. ISBN 978-3-540-33057-8. https://doi.org/10.1007/11691372_19.

[5] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. J. ACM, 58(6), 2011.https://doi.org/10.1145/2049697.2049700.

[6] M. Sighireanu et al. SL-COMP: Competition of Solvers for Separation Logic. In Proc. TACAS, pages 116–132. Springer International Publishing, 2019. ISBN 978-3-030-17502-3. https://doi.org/10.1007/978-3-030-17502-3_8.

[7] P. Muller and T.áš Vojnar. CPAlien: Shape Analyzer for CPAChecker. In Proc. TACAS, pages 395–397. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-54862-8. https://doi.org/10.1007/978-3-642-54862-8_28.

Appendix

Evaluation

- BenchExec tables:

file:///home/mo/Documents/Thesis/talk/benchexec/SLCPA_ModelSAT_CVC4.results.html

file:///home/mo/Documents/Thesis/talk/benchexec/SLCPA_ModelSAT_MathSAT5.results.html

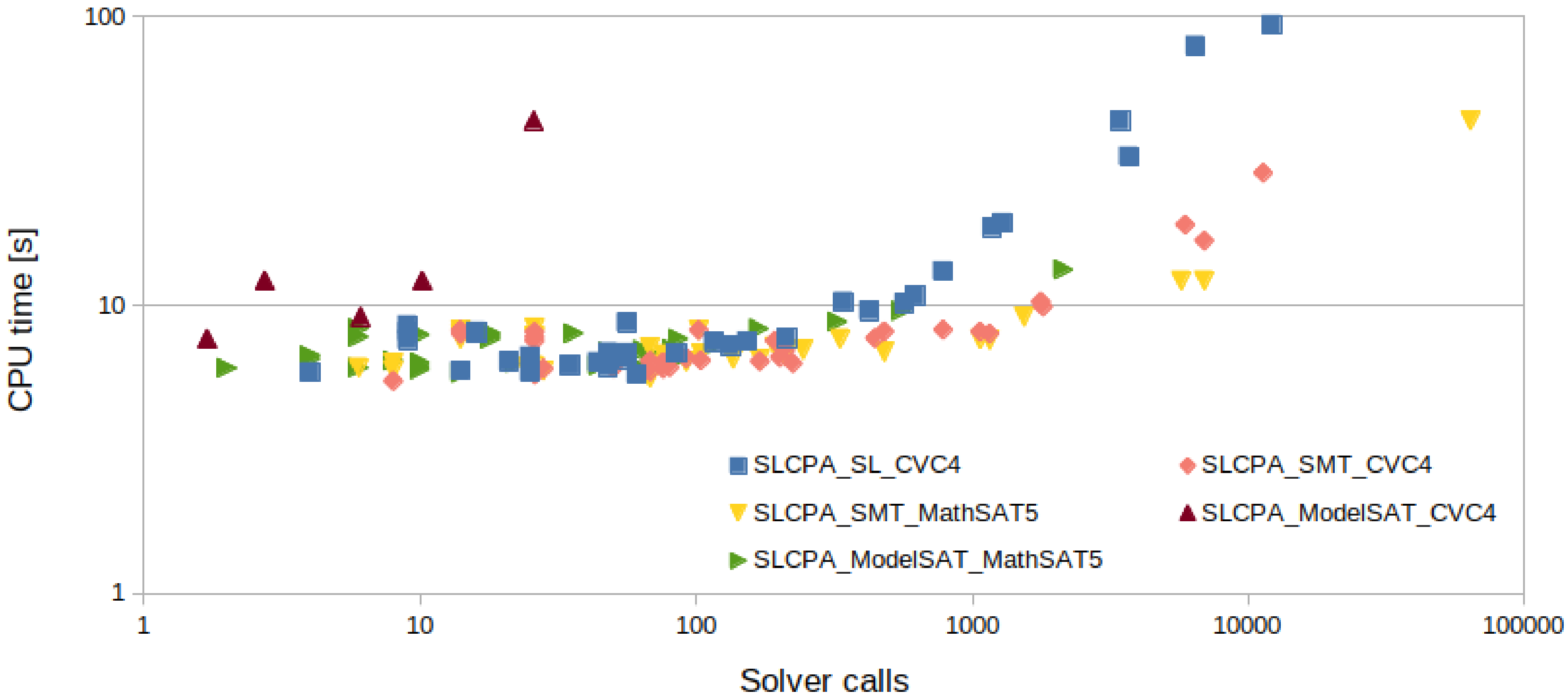
file:///home/mo/Documents/Thesis/talk/benchexec/SLCPA_SL_CVC4.results.html

file:///home/mo/Documents/Thesis/talk/benchexec/SLCPA_SMT_CVC4.html

file:///home/mo/Documents/Thesis/talk/benchexec/SLCPA_SMT_MathSAT5.html

file:///home/mo/Documents/Thesis/talk/benchexec/SMGCPA.results.html

Evaluation – Solver calls and CPU time



- CPU time increases with amount of solver calls
- SLCPA_SMT_CVC4 faster than SLCPA_ModelSAT_CVC4 although significantly more solver calls

Pointer Analysis – Locations and Values

$$\frac{\&x = SymLoc(x)}{\Sigma \models x \Downarrow_l \&x}\ \textsc{Var} \qquad \frac{\Sigma \models \epsilon \Downarrow_v E}{\Sigma \models *\epsilon \Downarrow_l E}\ \textsc{Ptr}$$

$$\frac{\Sigma \models x \Downarrow_l E \quad \Sigma \models \epsilon \Downarrow_v E' \quad F = E + E' \cdot sizeof(type(*x))}{\Sigma \models x[\epsilon] \Downarrow_l F}\ \textsc{Array}$$

$$\frac{\Sigma \models x \Downarrow_l E \quad F = E + Offset(x, y)}{\Sigma \models x.y \Downarrow_l F}\ \textsc{Field}$$

Operational semantics of $\Downarrow_l$

Pointer Analysis – Locations and Values

$$\frac{}{\Sigma \models \kappa \Downarrow_v \kappa}\ \textsc{Const} \qquad \frac{\Sigma \models \epsilon_\ell \Downarrow_l E \quad n = sizeof(\epsilon_\ell) \quad F = \Sigma[E]^n}{\Sigma \models \epsilon_\ell \Downarrow_v F}\ \textsc{Lhs}$$

$$\frac{\Sigma \models \epsilon_\ell \Downarrow_l E}{\Sigma \models \&\epsilon_\ell \Downarrow_v E}\ \textsc{AddressOf} \qquad \frac{\begin{array}{ll}\Sigma \models \epsilon_0 \Downarrow_v E & G = E \odot F \\ \Sigma \models \epsilon_1 \Downarrow_v F & type(\epsilon_0) = type(\epsilon_1)\end{array}}{\Sigma \models \epsilon_0 \odot \epsilon_1 \Downarrow_v G}\ \textsc{BinExp}$$

$$\frac{E = SymVal(f)}{\Sigma \models f(\epsilon_0, \ldots, \epsilon_n) \Downarrow_v E}\ \textsc{FunCall} \qquad \frac{\Sigma \models \epsilon \Downarrow_v E \quad F = cast(E, type(\epsilon), \tau)}{\Sigma \models (\tau)\epsilon \Downarrow_v F}\ \textsc{Cast}$$

$$\frac{\begin{array}{ll}\Sigma \models \epsilon_0 \Downarrow_v E & \epsilon_0 \text{ is a pointer} \\ \Sigma \models \epsilon_1 \Downarrow_v F & G = E \circledast (F \cdot sizeof(type(*\epsilon_0)))\end{array}}{\Sigma \models \epsilon_0 \circledast \epsilon_1 \Downarrow_v G}\ \textsc{PtrArithmetic}$$

Operational semantics of $\Downarrow_v$

Pointer Analysis – Transfer Relation

- Declaration edge:

$$\frac{\Sigma \models x \Downarrow_l E \quad n = sizeof(\tau)}{\{\Pi \wedge \Sigma\}\ \tau\ x\ \{\Pi \wedge \Sigma * E \mapsto -^0, \ldots, -^{n-1}\}}\ \textsc{Declare}$$

Pointer Analysis – Transfer Relation

- Statement edge: assignment

$$\frac{n = sizeof(\epsilon_\ell) \qquad \begin{array}{c}\Sigma * E \mapsto -^0, \ldots, -^{n-1} \models \epsilon_\ell \Downarrow_l E \\ \Sigma * E \mapsto -^0, \ldots, -^{n-1} \models \epsilon \Downarrow_v F\end{array}}{\{\Pi \wedge \Sigma * E \mapsto -^0, \ldots, -^{n-1}\}\ \epsilon_\ell = \epsilon\ \{\Pi \wedge \Sigma * E \mapsto F^0, \ldots, F^{n-1}\}}\ \textsc{Assign}$$

$$\frac{\Sigma \models \epsilon \Downarrow_v nil}{\{\Pi \wedge \Sigma\}\ \epsilon_\ell = \epsilon\ \{\perp^R\}}\ \textsc{Assign}^{\mathrm{InvR}} \qquad \frac{\Sigma \models \epsilon_\ell \Downarrow_l nil}{\{\Pi \wedge \Sigma\}\ \epsilon_\ell = \epsilon\ \{\perp^W\}}\ \textsc{Assign}^{\mathrm{InvW}}$$

Pointer Analysis – Transfer Relation

- Statement edge: dynamic allocation

$$\frac{}{\{\Pi \wedge \Sigma\}\ \texttt{malloc}(\epsilon)\ \{\bot^L\}}\ \textsc{Malloc}^{\text{Leak}}$$

$$\frac{\begin{array}{cc} \Sigma * E \mapsto -^0, \ldots, -^{n-1} \models \epsilon_\ell \Downarrow_l E & n = sizeof(\epsilon_\ell) \\ \Sigma * E \mapsto -^0, \ldots, -^{n-1} \models \epsilon \Downarrow_v \kappa & \Sigma \models \texttt{malloc}(\epsilon) \Downarrow_v x \end{array}}{\{\Pi \wedge \Sigma * E \mapsto -^0, \ldots, -^{n-1}\}\ \epsilon_\ell = \texttt{malloc}(\epsilon)\ \left\{ \begin{array}{c} \Pi \wedge \Sigma * E \mapsto x^0, \ldots, x^{n-1} \\ * \ x \mapsto -^0, \ldots, -^{\kappa-1} \end{array} \right\}}\ \textsc{Malloc}$$

Pointer Analysis – Transfer Relation

- Function call edge:
  - Allocated memory for parameters (Σ^P) and return value (Σ^R)

$$\Sigma^P(f, \{\epsilon_0, ..., \epsilon_{n-1}\}) := \bigstar_{i=0}^{n-1}(SymLoc(f_i) \mapsto F_i^0, ..., F_i^{sizeof(\epsilon_i)-1}))$$

$$\Sigma^R(f) := SymLoc(f) \mapsto \_^0, ..., \_^{sizeof(f)-1}$$

Pointer Analysis – Transfer Relation

- Function call edge:

$$\frac{type(f) = \texttt{void}}{\{\Pi \wedge \Sigma\}\ f(\epsilon_0, ..., \epsilon_n)\ \{\Pi \wedge \Sigma * \Sigma^P(f, \{\epsilon_0, ..., \epsilon_n\})\}}\ \textsc{FunCall}^{\text{void}}$$

$$\frac{\Sigma^f = \Sigma^P(f, \{\epsilon_0, ..., \epsilon_n\}) * \Sigma^R(f) \quad type(f) \neq \texttt{void}}{\{\Pi \wedge \Sigma\}\ f(\epsilon_0, ..., \epsilon_n)\ \{\Pi \wedge \Sigma * \Sigma^f\}}\ \textsc{FunCall}$$

$$\frac{\exists \epsilon \in \{\epsilon_0, ..., \epsilon_n\}.(\Sigma \models \epsilon \Downarrow_v nil)}{\{\Pi \wedge \Sigma\}\ f(\epsilon_0, ..., \epsilon_n)\ \{\bot^R\}}\ \textsc{FunCall}^{\text{Inv}}$$

Pointer Analysis – Transfer Relation

- Return statement edge:

$$\frac{type(f) = \texttt{void}}{\{\Pi \wedge \Sigma\}\ \texttt{return}_f\ \{\Pi \wedge \Sigma\}}\ \textsc{Return}^{\text{void}}$$

$$\frac{E = SymLoc(f) \quad \Sigma \models \epsilon \Downarrow_v F \quad type(f) \neq \texttt{void} \quad n = sizeof(f)}{\{\Pi \wedge \Sigma * E \mapsto -^0, \ldots, -^{n-1}\}\ \texttt{return}_f\ \epsilon\ \{\Pi \wedge \Sigma * E \mapsto F^0, \ldots, F^{n-1}\}}\ \textsc{Return}$$

Pointer Analysis – Transfer Relation

- Function return edge:

$$\frac{type(f) = \texttt{void}}{\{\Pi \wedge \Sigma * \Sigma^P(f, \{x_0, ..., x_n\})\}\ f(x_0, ..., x_n)\ \{\Pi \wedge \Sigma\}}\ \textsc{FunRet}^{\text{void}}$$

$$\frac{type(f) \neq \texttt{void}}{\{\Pi \wedge \Sigma * \Sigma^P(f, \{x_0, ..., x_n\}) * \Sigma^R(f)\}\ f(x_0, ..., x_n)\ \{\Pi \wedge \Sigma\}}\ \textsc{FunRet}$$

Pointer Analysis – Transfer Relation

- Variable Scope:

$$\frac{\Sigma \models x \Downarrow_l E \qquad n = sizeof(x) \qquad \Sigma \models x \Downarrow_v F \qquad \bigwedge_{i=0}^{n-1} Leak(\Sigma, F^i) = true}{\{\Pi \land \Sigma * E \mapsto F^0, ...F^{n-1}\}\ oos(x)\ \{\bot^L\}} \ \textsc{OutOfScope}^{\text{Leak}}$$

  - special statement `oos(x)` representing a variable `x` that goes out of scope

Implementation

- SL State

$$
\begin{aligned}
\Sigma^s &: \texttt{LinkedHashMap}\langle\texttt{BVFormula} \rightarrow \texttt{BVFormula}\rangle \\
\Sigma^h &: \texttt{LinkedHashMap}\langle\texttt{BVFormula} \rightarrow \texttt{BVFormula}\rangle \\
\Pi &: \texttt{BooleanFormula} \\
SegmentSizes &: \texttt{Map}\langle\texttt{BVFormula} \rightarrow \texttt{Int}\rangle \\
Allocas &: \texttt{Map}\langle\texttt{String} \rightarrow \texttt{Set}\langle\texttt{BVFormula}\rangle\rangle \\
SSA\text{-}Indices &: \texttt{SSAMap} \\
Properties &: \texttt{Set}\langle\texttt{String}\rangle
\end{aligned}
$$

Implementation

- Why SSAMap?

```
int main() {
  char *p;
  for(char i=0; i<2; i++) {
    char x;
    if(i==0) {
      p = &x;
    } else {
      *p = 1;
    }
  }
  return 0;
}
```

→ SymLoc(x) = &main:x@i

Implementation

- Dereferencing with model generation:

Input: an expression E as `BVFormula` and a `Map` M describing a symbolic heap
Output: a pair of `BVFormula` denoting the location and value of the cell at address E if allocated, nil otherwise

if $M.containsKey(E)$ **then**
  return $M.get(E)$;
end
`// Assume auxiliary variable` aux_k `for each` k
$formula = \bigwedge_{k \in M.keys()} k = E \iff aux_k$;
if $SAT(formula)$ **then**
  $model = getModel(formula)$;
  foreach $(aux_k, value) \in model$ **do**
    if $value$ **then**
      return k;
    end
  end
end
return nil;