LMU Munich
Faculty of Mathematics, Informatics and Statistics
Institute for Informatics

Master's Thesis

Solver-based Analysis of Memory Safety using Separation Logic

submitted by:
Moritz Beck

Advisor:
Martin Spießl

Supervisor:
Prof. Dr. Dirk Beyer

31st August, 2020

LMU Munich
Institute for Informatics
Oettingenstraße 67
80538 Munich
Germany

Urheberschaftserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement of Authorship

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Munich, 31st August, 2020

Acknowledgements

First of all, I would like to thank Martin for the great supervision during the whole thesis. You never hesitated to share your expertise with me and the work with you improved my skills a lot. Further, I give thanks to my family who has supported me in the entire period of time - thereby, a special thanks to my investors. Last but not least, I want to thank my better half Julia. You always encouraged me and have never been doubtful about my success. Morulia^{07}.

Abstract

As the amount and complexity of safety-involved systems is continuously rising, automatic software verification gains more and more in importance. In this concern, Separation Logic has proved to be a promising way to cope with problems related to automation and scalability.

Some SMT solvers provide support for Separation Logic, and there even is a competition on Separation Logic (SL-COMP) which aims at standardizing the way to formulate Separation Logic problems in a common format (SMT-LIB). In this spirit, this thesis explores the viability of using a Separation Logic solver in the context of software verification. We do so by investigating the requirements of a suitable solver interface in the scope of a pointer analysis based on symbolic execution.

Contents

List of Figures **vi**

List of Acronyms **vii**

1 Introduction **1**

2 Related Work **4**

3 Background **6**

3.1 Hoare Logic 6

3.2 Separation Logic 6

3.2.1 Frame Inference and Abduction 7

3.3 Satisfiability Modulo Theories 8

3.4 Configurable Program Analysis 9

3.4.1 CPA algorithm 10

4 Theory **12**

4.1 Abstract Domain 12

4.2 Pointer Analysis 14

4.2.1 Memory Model 14

4.2.2 Language 15

4.2.3 Memory Access 16

4.2.4 Locations and Values 17

4.2.5 Transfer Relation 20

5 Implementation **25**
5.1 Symbolic Heap Formula 25
5.2 SL State 26
5.3 Solver 27

6 Evaluation **31**
6.1 Execution Environment 31
6.2 Results 33

7 Discussion **34**

Appendix A **36**
A.1 Reachability check with cycles 36
A.2 SLCPA Configuration - sl.properties 36
A.3 BenchExec - integration-sl.xml 37

Bibliography **38**

List of Figures

1.1 An example C program (1.1a) with its corresponding control-flow automaton (1.1b) 2

4.1 Grammar of formulae describing a symbolic heap 13
4.2 Three variables with different types separately in memory 14
4.3 A subset of C 15
4.4 The operational semantics of transforming an expression to its corresponding location (4.4a) and value (4.4b) 18

5.1 CPA structure 25
5.2 The SL state 26

6.1 The elapsed CPU time (6.1a) and used memory (6.1b) as well as the combination of both in 6.1c 32
6.2 Assignment of CPU time to solvertime (red) and other computations (green) 33

List of Acronyms

SL Separation Logic 1

CPA Configurable Program Analysis 5

CFA Control Flow Automaton 2

SMT Satisfiability Modulo Theories 8

SMG Symbolic Memory Graph 4

1 Introduction

Automatic software verification is and has always been a trade-off between precision and expense [5]. For very complex systems with a huge amount of code, the efficiency of a method is crucial for its practical relevance. Especially in the field of shape analysis, many procedures are limited in their application due to a lack of computational resources.

This is where *Separation Logic* (SL) comes into play. Pym et al. state two reasons: "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 [...]" [18]. In doing so, SL provides an intuitive representation of (allocated) memory by fragmenting the program's heap into smaller, independent heaplets. In this fashion, the theory comes with a versatile expressiveness that allows the modeling of all kinds of data structures.

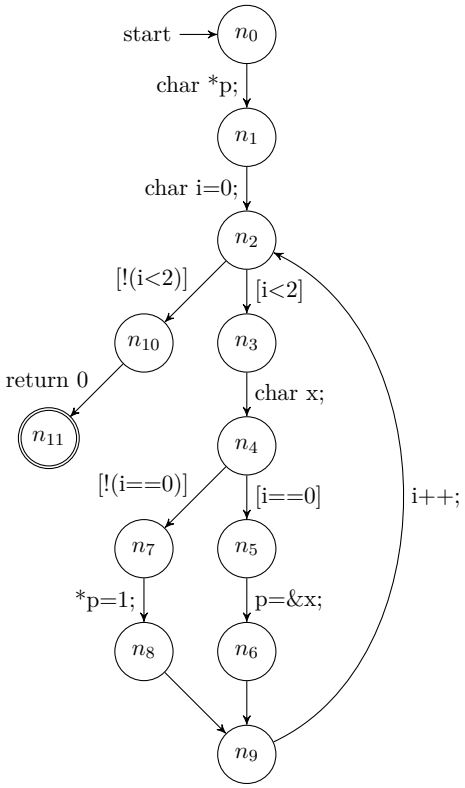
The ensuing potential is already attested as it became part of software verification tools targeting practice-oriented problems [10, 22]. Further, the SL-COMP builds a baseline of applied SL. Initiated in the year 2014 and lastly organized in 2019, SL-COMP established a standard for the formalization of SL-problems based on SMT-LIB [21]. Thereby, the extension of SMT-LIB by the spatial predicates of SL opens up new dimensions in respect to automated reasoning.

In the scope of this thesis, we want to assess the above mentioned capabilities with a pointer analysis considering C programs. For each line of code, we derive the associated heap manipulation and represent it as SL-formula. We construct and analyze these formulae according to memory safety properties with the use of JavaSMT, an interface to various SMT solvers. By embedding the analysis into the CPAchecker framework, we benefit from concepts of both model checking and program analysis.

Motivating Example

Figure 1.1 shows a small C program (a) and its corresponding *Control Flow Automaton* (CFA) (b). The CFA - formally discussed in chapter 3 - models the program as a transition graph with edges labeled with the statements of the program. The code of the example contains an invalid pointer assignment inside

```
1   int main() {
2     char *p;
3     for(char i=0; i<2; i++) {
4       char x;
5       if(i==0) {
6         p = &x;
7       } else {
8         *p = 1;
9       }
10    }
11    return 0;
12  }
13
```



(a) *Unsafe program* ***(b)*** *Control-flow automaton*

Figure 1.1: *An example C program (1.1a) with its corresponding control-flow automaton (1.1b)*

the for-loop. Since the memory location associated to the local variable `x` is automatically (re)allocated for each loop iteration, the variable `p` points to an already disposed address at the time of dereferencing in line 8. However, the null pointer cannot be easily detected by a compiler, as the memory segment is indirectly accessed.

Our approach targets this issue among others by providing a SL-formula for each state along the execution path. For the assignment edge in line 8, this brings us to the simplified formula

$$p \mapsto x_0 * i \mapsto 1 * x_1 \mapsto 0$$

whereby the indices of x_i indicate different locations for x. One reads the formula as "p points to x_0 and separately i points to 1 and separately x_1 points to 0". This wording implies, that $p \neq i \neq x_1$ must hold. We use this characteristic to handle allocation checks as follows:

As the value x_0 of the memory cell at address p is not allocated on the heap, the invalid dereference of the assignment $*p = 1$ should be determined. In order to do so, the previous formula is supplemented by another heaplet $x_0 \mapsto v$ with v being a free variable. This procedure constitutes a crucial part of our approach, since the determination of the heaplet requires the resolution of the pointer expression $*p$ to its value x_0. The resulting formula is eventually passed to a solver supporting SL:

$$p \mapsto x_0 * i \mapsto 1 * x_1 \mapsto 0 * x_0 \mapsto v$$

The formula is satisfied for any variable assignment that holds $p \neq i \neq x_1 \neq x_0$. And in fact there are infinite trivial solutions. By implication, if x_0 was indeed allocated on the heap the following formula is constructed:

$$p \mapsto x_0 * i \mapsto 1 * x_1 \mapsto 0 * x_0 \mapsto v * x_0 \mapsto v'$$

This results in the constraint $x_0 \neq x_0$ which is unsatisfiable by definition. As a consequence, we acquire an allocation check by the negation of the satisfiability check of the constructed heap formula.

The discussed example outlines the base frame of our approach. In order to clarify the negotiated challenges, the thesis is structured as follows: At first, related work to applied SL and pointer analysis in general is discussed to substantiate the potential of SL theory while simultaneously emphasizing the benefits of this work. Afterwards, a deeper understanding of the underlying theory and related technologies is given. Chapter 4 constitutes the main part of the thesis. Here, the heap formula construction and the associated pointer analysis is concretized. Subsequently, important implementation details are outlined and eventually, the results of the approach are presented with a prospect to future extensions.

2 Related Work

Automated reasoning has become a very important topic in the field of computer science. Thereby, it is mandatory to analyze code bases at a certain scale to achieve a practical relevance. As a consequence, there arise huge challenges considering computational resources. SL targets the problem of scalability by composition. Further, its practical application has already been approved by existing approaches [9, 10].

One of the first shape analysis build on SL was initiated by the tool called Space Invader [8]. As an adaptive analysis, Space Invader provides lists of SL formulae to describe a program's heap. Providing a join-operator as well, the tool is capable of handling programs with more than one thousand lines of code. Nevertheless, the manual adaption of higher-order predicates is time-consuming [8]. In this respect, the tool was further extended by a compositional approach described in [7]. The compositionality is achieved by "[...] inferring a precondition and postcondition for a procedure, without knowing its calling context [...]" [8]. The resulting version was finally published called Abductor in the scope of [9]. To the best of our knowledge, Facebook Infer might be one of the latest tools of applied SL. It is a static software analyzer for multiple program languages and uses bi-abduction techniques to spot errors caused by null pointer access or memory leaks [10, 18].

Another approach of a shape analysis depicts the so called *Symbolic Memory Graph* (SMG) [12, 13]. Initially inspired by SL, in particular Space Invader, the abstract domain of a SMG characterizes a state by providing a graph representation of the heap. The concept was implemented for the first time by the Predator tool [12] and there is also a related implementation as a part of CPAchecker [4] called CPAlien [15]. As the graphs are strongly optimized for the analysis of lists, such as those used in the Linux kernel, the application of SMGs in terms of other data structures than lists may be quite involved. However, in case of CPAlien the integration into CPAchecker made the concept of SMGs convenient to use.

The CPAchecker framework realizes the reachability algorithm of the *Configurable Program Analysis* (CPA) [5]. It accumulates a wide range of concepts in the area of both, model checking and program analysis. Thereby, CPAchecker additionally provides parser capabilities (including C) and further grants access to JavaSMT [4, 15], an interface for SMT solvers written in Java. Besides, JavaSMT already supports SL solvers that address problems defined in the scope of SL-COMP. As this thesis targets the practicability of such SL solvers, the described infrastructure constitutes an important tool of our approach.

3 Background

SL is able to give insights about physical resources a program (or one of its subroutines) draws from. Together with its ability to cope with scalability, SL has high potential to close the gap between formal theory and practice-oriented, *automated* software verification. Concerning the latter, CPA provides a promising concept in order to assess the practicability of Separation Logic in a systematic and flexible way. In this chapter, the underlying theories as well as their formal concepts are described.

3.1 Hoare Logic

Reasoning about programs is always reasoning about state. Hoare Logic [14] addresses this concern by inventing the *Hoare Triple*, which describes how a certain piece of code alters the program's state. In the form of

$$\{P\}C\{Q\}$$

it defines how the *precondition* P is transferred to the *postcondition* Q executing the *command* C. By formulating rules of inference, Hoare Logic enables formal proofs of program properties. Therewith, it has become very important in the area of software verification, not only in theoretical but also in practice-oriented respect.

3.2 Separation Logic

In the early 2000's J. Reynolds and P. O'Hearn et al. extended the classical Hoare Logic to the theory of SL [16, 20] building the foundation of *Symbolic Heaps* [2, 11]. Thereby, the program's state is defined by two components, a *pure* and a *spatial* part, respectively a *store* and a *heap*. The store keeps track of the variables by assigning them to values whereas the heap represents the

allocated memory cells as locations mapped to values using the *points-to-relation* $\mapsto$ [11, 16]. Together with the *separating conjunction* $*$ (star) a heap can be segmented into chunks each to be considered separately. The spatial connective *emp* denotes the empty heap. As an example, the following logical formula denotes a cyclic list with two allocated nodes:

$$x \mapsto y \ * \ y \mapsto x$$

In contrast to the and-conjunction $\wedge$ - as part of Hoare Logic - $*$ clarifies that here x and y are "separately in memory" and thus no aliases. This characteristic stands in direct relation to the *Frame Rule*

$$\frac{\{P\}C\{Q\}}{\{P * F\}C\{Q * F\}} \qquad \text{(Frame Rule)}$$

saying, that if the precondition P suffices for the program C to run leading to the postcondition Q, there can be additional allocated memory remaining unaffected (the frame F). The Frame Rule holds if C does not modify any free variables included in F: $Modifies(C) \cap Free(F) = \emptyset$ [11, 16, 18]. "This support for local reasoning is critical, supporting compositional reasoning about large programs by facilitating their decomposition into many smaller programs that can be analysed and verified independently." [18]

3.2.1 Frame Inference and Abduction

The idea of compositional reasoning raises the question how to decide whether a certain part of memory is or is not affected by a program. Hence, the primary goal is to determine the frame F to be as precise as possible. Let us assume - concerning the previous example of a cyclic list - a symbolic heap P entails the necessary memory cells:

$$P \ \vdash \ x \mapsto y \ * \ y \mapsto x$$

If now x points to a new value z, only the first part of the heap formula will be affected. Thus the frame F can be determined as $y \mapsto x$ (cf. Frame Rule).

Although this frame inference is exposed to be trivial, it becomes arbitrarily complex for large programs. An efficient method to address this issue and determine the frame in a prover-based way is called *bi-abduction* [9].

$$A * ?antiframe \vdash B * ?frame \qquad \text{(Bi-Abduction)}$$

Given two symbolic heaps A and B, bi-abduction (see algorithm 1) discovers a pair of *frame* and *antiframe*. The latter is the part of memory that is affected by

Algorithm 1: Bi-abduction from Calcagno et al. [9]

Input: symbolic heaps A and B
Output: $antiframe$, $frame$

$antiframe = Abduce(A, B) \implies A * antiframe \vdash B$
$frame = Frame(A * antiframe, B) \implies A * antiframe \vdash B * frame$
return *(antiframe, frame)*

a program (*footprint*), whereas the frame remains unchanged. The computation is done in two distinct steps: First, the antiframe is inferred recursively using ranked abduction rules. Consequently, a set of potential solutions is acquired and a suitable "best" is chosen in a heuristic manner. Nevertheless, abduction is about forming hypotheses (cf. Peirce, p. 106 [17]). In other words the assertion of - potentially weakened - preconditions might be unsound. This is why the algorithm's second step is crucial for its correctness.

As we have seen, SL is able to give a formal representation of heap-manipulating programs. The related logical formulae are composed of two distinct parts. The store contains all of classic logic and boolean expressions, whereas the heap additionally includes the spatial connectives. Thus, it needs special solvers to deal with this kind of formula. The next section attends to this matter and gives a broad overview of the underlying concepts and the steadily growing community.

3.3 Satisfiability Modulo Theories

The decision problem of *Satisfiability Modulo Theories* (SMT) targets the satisfiability of first-order logic formulae considering a chosen theory. For this purpose, a SMT solver is used to determine for a given formula whether at least one solution exists or not. SMT solvers are strongly related to SAT solvers. Although the latter are limited to propositional formulae, SMT solvers operate on a higher abstraction level. This allows them to gain in performance compared to pure SAT solvers due to the implementation of theories like arrays and bit-vectors. In this way, an encoding overhead to the bit level is avoided [1].

The high impact of SMT solvers in the area of automated reasoning lets the community rise in a continuous manner. As a result, standards and a wide range of related projects arose.

SMT-LIB constitutes the approach to provide a common input language to formulate SMT problems. SMT-LIB includes a standard format for SMT solvers as well as standard definitions of background theories. Furthermore a set of

benchmarks facilitates to analyze and improve new approaches [1]. In this spirit, the annual SMT-COMP organizes a competition for SMT solvers since 2005 in order to boost the state of the art in SMT.

SL-COMP - as a counterpart to SMT-COMP - provides an analogous competition for solvers supporting SL. The input format is also based on SMT-LIB. With eleven competitors in the last edition of the competition, SL-COMP has already contributed to the SL community to a significant degree [21].

3.4 Configurable Program Analysis

Beyer et al. introduced the concept of CPA that brings both model checking and program analysis together [5]. On the one hand, model checking is traditionally path-sensitive. Thus, it can result in an overwhelming or even infinite growth of the abstract reachability tree for large programs. On the other hand, program analyses most commonly join program paths at the expense of precision [5].

In their formalism, Beyer et al. make use of a CFA $CFA = (L, pc_0, G)$ as the semantic representation of a program. The set of program locations L represents the program counter pc with pc_0 as the initial location, meaning the program entry. The set $G \subseteq L \times Ops \times L$ depicts the control-flow edges between program locations with Ops as the set of supported operations. If such an operation is executed, the program counter and other variables are modified. The assignment of values to all variables in $X \cup \{pc\}$ is referred to as the program's concrete state c with C as the set of all concrete states. Each control-flow edge $g \in G$ defines a labeled transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$. Assumed a concrete state c and its successor c', there exists a control-flow edge $g \in G$ with $(c, g, c') \in \xrightarrow{g}$ or $c \xrightarrow{g} c'$ for short.

The CPA-formalism implements a reachability analysis in a practice-oriented, experimental fashion. It composes different (independent) modules - the CPAs - that can be adjusted to the individual working task concerning precision and performance [5]. Each CPA $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$ consists of the following components defined by Beyer et al. [5]:

1. The *Abstract Domain* $D = (C, \mathcal{E}, [\![\cdot]\!])$ represents the states a program can occupy during execution. A distinction is made between the concrete states C and the elements of the semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$, the abstract states E. The linkage between the two is captured due to the concretization function $[\![\cdot]\!] : E \longrightarrow 2^C$, that maps each abstract state to a set of concrete ones. Further, the semi-lattice $\mathcal{E}$ defines a preorder $\sqsubseteq \subseteq E \times E$, a join-

operator $\sqcup : E \times E \longrightarrow E$ as well as a least upper bound $\top \in E$ and a greatest lower bound $\bot \in E$.

2. The *Transfer Relation* $\rightsquigarrow \subseteq E \times G \times E$ provides the successors for each abstract state. Analogous to the transition relation over concrete states, each transfer is labeled with a control-flow edge. Thus, for an abstract state e and a possible successor e' there exists a control flow edge $g \in G$ with $e \overset{g}{\rightsquigarrow} e'$

3. "The *Merge-Operator merge* $: E \times E \longrightarrow E$ combines the information of two abstract states. To guarantee soundness of [...][the] analysis" [5] $e' \sqsubseteq merge(e, e')$ has to be fulfilled. In our approach we use $merge^{sep}(e, e') = e'$.

4. The *Termination-Check stop* $: E \times 2^E \longrightarrow \mathbb{B}$ determines for each new encountered abstract state along the reachability analysis, whether it is coped by the already visited states or not. Hence, it ensures the termination of the CPA-algorithm.

3.4.1 CPA algorithm

The reachability algorithm $CPA(\mathbb{D}, e_0)$ (see algorithm 2) returns for a given configurable program analysis $\mathbb{D}$ and an initial state e_0 a set *reached* of all abstract states that have been encountered. Together with *reached*, the algorithm updates another set *waitlist* of abstract states that have to be treated. The algorithm terminates when *waitlist* is empty.

At the beginning, both sets have one entry, the initial abstract state e_0. For all elements in *waitlist* the algorithm determines its successors obtained from the transfer relation $\rightsquigarrow$. Now, each abstract successor state e' is merged with each already encountered abstract state e'' in *reached*. In the case that information was added to e'', i.e. $merge(e', e'') \neq e''$, e'' is replaced by the merge result. If the abstract successor state is not yet covered by *reached* after the merge-step, it is added to *waitlist* and *reached* to be processed later on.

In this chapter we illustrated the bases of our approach. One of the main challenges in the field of software verification is the trade-off between precision and performance. As discussed earlier, SL is able to bring both properties together. The combination with the concept of CPA allows us to investigate its capabilities in the scope of a pointer analysis.

Algorithm 2: $CPA(\mathbb{D}, e_0)$ from Beyer et al. [5]

Input: a configurable program analysis $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$, an initial abstract state $e_0 \in E$, let E denote the set of elements of the semi-lattice of D
Output: a set of reachable abstract states
Data: a set *reached* of elements of E, a set *waitlist* of elements of E

$waitlist := \{e_0\}$
$reached := \{e_0\}$
while $waitlist \neq \emptyset$ **do**
  pop e from $waitlist$
  foreach e' *with* $e \rightsquigarrow e'$ **do**
    foreach $e'' \in reached$ **do**
      `// Combine with existing abstract state.`
      $e_{new} := merge(e', e'')$
      if $e_{new} \neq e''$ **then**
        $waitlist := (waitlist \cup \{e_{new}\}) \setminus \{e''\}$
        $reached := (reached \cup \{e_{new}\}) \setminus \{e''\}$
      end
    end
    if $\neg stop(e', reached)$ **then**
      $waitlist := waitlist \cup \{e'\}$
      $reached := reached \cup \{e'\}$
    end
  end
end
return *reached*

4 Theory

Our approach implements a pointer-analysis based on SL by following the concept of symbolic execution. We derive heap formulae from standard C code and analyze them in respect to memory safety properties. Thereby, we make use of the CPA formalism to gain from its extensive infrastructure of tools and concepts. In this chapter, we want to give an understanding of our method's structure and the semantics it encapsulates.

4.1 Abstract Domain

For each element of the abstract domain, a Hoare Triple $\{P\}C\{Q\}$ defines the program's abstract state. The command C is obtained from the CFA, that provides the related control-flow edge for each abstract state. The pre- and postcondition P and Q are represented by SL-formulae. These describe the program's state as a symbolic heap before and after C is executed.

$$\begin{aligned} \text{Vars} &:= \{\text{x, y, ...}\} & \text{Values} &:= \{\text{... -1, 0, 1, ...}\} \cup \{\text{nil}\} \\ \text{Vars'} &:= \{\text{x', y', ...}\} & \text{Locations} &\subseteq \text{Values} \\ \text{Stores} &:= (\text{Vars} \cup \text{Vars'}) \to \text{Values} & \text{Heaps} &:= \text{Locations} \rightharpoonup_{fin} \text{Values} \\ \text{States} &:= \text{Stores} \times \text{Heaps} \end{aligned}$$

According to [16] and [11], each state consists of a store and a heap. The store maps program variables $Vars$ and symbolic variables $Vars'$ to values. Elements of $Vars'$ do not occur in programs and are limited to logical formulae. The heap is a finite partial function from locations to values. In this concern, the satisfaction relation

$$s, h \models \Pi \wedge \Sigma$$

declares that a given store $s \in Stores$ and a (concrete) heap $h \in Heaps$ satisfy a symbolic heap with the *pure* part Π and *spatial* part Σ. In this respect, the

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

Figure 4.1: *Grammar of formulae describing a symbolic heap*

notation $\Sigma * P$ stands for the (disjoint) union of a formula P onto the spatial part of a symbolic heap, and respectively $\Pi \wedge P$ for the pure part.

Figure 4.1 shows the grammar of formulae representing a symbolic heap. The pure formulae describe the store by conjunctions of constraints that include relations among expressions. These expressions are limited by the theory of integers and are either constants, variables or arithmetic expressions involving all of these. The spatial formulae specify properties of the heap by assigning values to each memory cell. A single *points-to* predicate $x \mapsto y$ represents a heap with only one allocated cell at address E with content F. We use $E \mapsto -$ to indicate an arbitrary value for F. Besides, the following definition is used to declare a contiguous segment of allocated cells with their associated content:

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

Memory Safety Properties As defined by the CPA-formalism, the abstract domain specifies the greatest lower bound $\bot$. We define four abstract bottom states related to the memory safety properties that our pointer analysis targets:

- invalid read $\bot^R$,
- invalid write $\bot^W$,
- invalid free $\bot^F$ and
- memory leak $\bot^L$.

In order to facilitate a pointer analysis, SL-formulae as semantic representation of programs have to be constructed. We describe this procedure in the following section.

4.2 Pointer Analysis

The analysis creates a memory model for each node along the execution path. Every atomic command is transferred into its representative memory manipulation. Thus, we gain a SL-formula for each state that can be analyzed by a solver. Thereby, the following memory safety properties are observed: invalid dereference, further distinguished between invalid read and write, invalid free and memory leak.

4.2.1 Memory Model

In C (e.g. ISO/IEC 9899:1990) the memory is seen as a sequence of bytes, although the interpretation of a chunk of memory depends on the type of the pointer that is used for access. According to this, we model the symbolic heap Σ as a collection of byte-sequences, each byte having an unique address. In terms of SL, Σ is composed of locations equal to the size of *`void` pointing to values with the size of `char`.

For a variable $x \in Var$ (with a type greater than one byte) its assigned value is split into a sequence of adjacent cells. The size of the sequence corresponds to the type size and is determined using the function $sizeof(x)$ related to the `sizeof()` operator in C. Thereby, we define an injective function

$$SymLoc := Var \hookrightarrow Var'$$

that gives exactly one match $\&x \in Var'$ for each x.[1] The symbolic variable indicates the start of the segment as the memory address of x. Each consecutive location is then composed of $\&x$ and an offset. Figure 4.2 illustrates this procedure by the example of three variables with different types. The corresponding

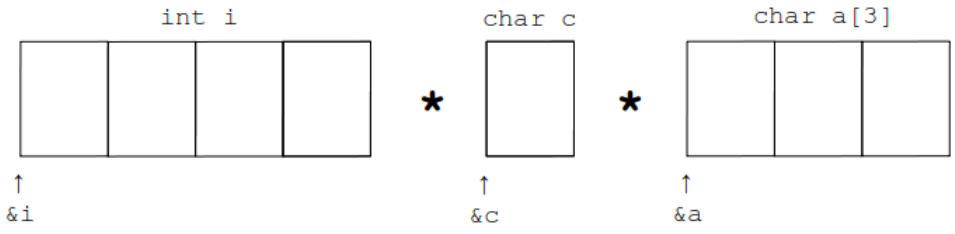


Figure 4.2: *Three variables with different types separately in memory*

[1] We use the notation &x to indicate the relation to the address-of operator in C.

SL-formula describes the symbolic heap as follows,

$$\&i \mapsto i^0, i^1, i^2, i^3 \ * \ \&c \mapsto c \ * \ \&a \mapsto a[0], a[1], a[2]$$

whereby the superscripts i^n indicate the n-th byte of the value i.

Another characteristic of the C memory management is the differentiation between several allocation methods, namely: static and global as well as automatic and dynamic allocation. The different memory sections are often referred to as *stack* and *heap*. We handle the related properties by subdividing Σ into two distinct parts:

$$\Sigma = \Sigma^s * \Sigma^h$$

Σ^h denotes the part of memory, that is allocated dynamically on runtime, for example due to `malloc()`. Everything else is then described by Σ^s. Therewith, we are able to detect invalid frees on the one hand, and apply solver calls more efficiently on the other hand.

4.2.2 Language

Our formalism copes with a subset of the standard C language illustrated in figure 4.3. A program is defined as a list of statements s. We essentially distinguish between declarations and assignments, as the CFA already handles more complex statements related to control flow and variable scope. The associated edges are illustrated in section 4.2.5. Expressions of type *CExpression* are divided into left-hand side expressions ϵ_ℓ and right-hand side expressions ϵ, whereby all of them exclude floats. At this point, we want to emphasize the distinction

$$\begin{aligned}
\tau &:= \text{typical types excluding floats} \\
\kappa &:= \text{typical constants} \\
\epsilon_\ell &:= x, y, ... \mid \ *\epsilon \mid a[\epsilon] \mid a.b \\
\epsilon &:= \kappa \mid \epsilon_\ell \mid \&\epsilon_\ell \mid \epsilon \odot \epsilon \mid f(\epsilon_r, ..., \epsilon_r) \mid (\tau)\epsilon \\
s &:= \tau \ a \mid \epsilon \mid \epsilon_\ell = \epsilon \mid \{s\} \mid s; s \\
&\quad \mid while(\epsilon) \ s \mid if(\epsilon) \ s \ else \ s
\end{aligned}$$

Figure 4.3: *A subset of C*

between expressions of the program (ϵ of type *CExpression*) and those that are part of the SL-formulae (E of type *Expression*). The left-hand side expressions include identifier, pointer and array subscript expressions as well as field references according to structures and unions. The right-hand side expressions

extend ϵ_ℓ by constants, the address-of operator, function calls and cast as well as binary expressions. Concerning the latter, the associated operators denoted by $\odot$ and constants κ refer to typical binary operators, respectively constants as part of the SL-formulae. Further, there exists a canonical mapping between identifier expressions and the elements of Var. For the sake of completeness, we make the following assumptions:

- A variable is declared only once. With this, a static typing of all variables is ensured. The $CType$ of an expression is determined using the function $type := CExpression \rightarrow CType$;
- The function $cast := Expression \times CType \times CType \rightarrow Expression$ abstracts from the concrete cast implementation of the target system;
- All binary operators of $\odot$ are solely defined for operands of the same type;
- We cope with pointer arithmetic by introducing the commutative operators $\circledast \subset \odot$ that allow addition and subtraction of pointer offsets.

4.2.3 Memory Access

Whenever a value is read from or written to memory, the associated location has to be dereferenced. Thereby, it is crucial to determine whether the location in form of an expression E points to an allocated cell on the heap Σ. We define

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

saying that a heap Σ satisfies the allocation of memory at address F that is semantically equivalent to address E.

It is crucial for the analysis to access the value F in some way. Assume a dereference function $Deref()$ that retrieves the value at a location E in a context heap Σ. If the location is invalid or if it is nil itself, the function returns nil.

$$Deref := Spatial\ formula \times Expression \rightarrow Expression$$

Here, we give a recursive definition that checks each heaplet whether it gives a model for $Allocated(E)$:

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

We use the function $Deref()$ in order to resolve memory accesses. A concrete implementation is given in chapter 5. However, in C a memory access also depends on the type of the dereferenced pointer. We target this characteristic with the use of the definition

$$\Sigma[E]^n := Deref(\Sigma, E) :: Deref(\Sigma, E+1) :: ... :: Deref(\Sigma, E+n-1)$$

denoting that the memory segment with size n starting at address E is dereferenced. By doing so, all bytes inside the segment are dereferenced and concatenated afterwards. If any of the memory accesses fails, *nil* is returned. The arrangement of the bytes depends on the endianness of the source code's target system. Here, the notation describes little-endian order.

4.2.4 Locations and Values

The pointer analysis captures the meaning of each $CExpression$ by transforming it to a primitive $Expression$ as part of the SL-formulae defined in 4.1. Thereby, a distinction is made between locations and values as the interpretation of C code implies. For example, an assignment in C is composed of a left- and right-hand side $\epsilon_\ell = \epsilon$. The meaning of this assignment can be worded as "the value of ϵ is written to the location of ϵ_ℓ". It is crucial that expressions referring to the same memory location are transformed to the same symbolic representation. Assume two variables `char x` and its pointer `char* p = &x`. Then, all of the following expressions refer to the same memory location: `x`, `x[0]` and `*(p + x - 1)`. In this spirit, we introduce two notations:

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

The first denotes the evaluation of a left-hand side expression ϵ_ℓ to its corresponding *location*, whereas the second retrieves the *value* of ϵ. Both evaluations are accomplished in the context of a symbolic heap Σ. Figure 4.4 provides the corresponding semantic rules.

As the definition of $\Downarrow_l$ implies, only left-hand side expressions can be evaluated to a location. For the trivial case of identifier expressions representing program variables $x \in Var$, $SymLoc()$ provides an unique symbolic location for each variable. For array subscript expressions in the form of $a[\epsilon]$, the location of a is incremented by the evaluated offset, meaning the value of ϵ. Similar to the procedure of a compiler, a factor according to the type of a is multiplied to the value. Pointer expressions $*\epsilon$ are similarly treated since they can be rewritten as array subscripts. Their evaluation is straightforward by evaluating the operand.

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

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

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

a: *The operational semantics of* $\Downarrow_l$

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

$$\frac{\Sigma \models \epsilon_\ell \Downarrow_l E}{\Sigma \models \&\epsilon_\ell \Downarrow_v E}\ \textsc{AddressOf} \qquad \frac{E = SymVal(f)}{\Sigma \models f(\epsilon_0, ..., \epsilon_n) \Downarrow_v E}\ \textsc{FunCall}$$

$$\frac{\Sigma \models \epsilon \Downarrow_v E \quad F = cast(E, type(\epsilon), \tau)}{\Sigma \models (\tau)\epsilon \Downarrow_v F}\ \textsc{Cast}$$

$$\frac{\begin{array}{cc}\Sigma \models \epsilon_0 \Downarrow_v E & G = E \odot F \\ \Sigma \models \epsilon_1 \Downarrow_v F & type(\epsilon_0) = type(\epsilon_1)\end{array}}{\Sigma \models \epsilon_0 \odot \epsilon_1 \Downarrow_v G}\ \textsc{BinExp}$$

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

b: *The operational semantics of* $\Downarrow_v$

Figure 4.4: *The operational semantics of transforming an expression to its corresponding location (4.4a) and value (4.4b)*

In order to handle composite types such as structures and unions, the associated field accesses have to be exchanged by location offsets. We introduce the offset function $Offset(x, m)$.

$$Offset := CExpression \times CExpression \to Int$$

It returns for a variable x the offset of its member m including potential padding. In the case of unions, the offset is always equal to zero. For structures the offset function traverses the member list of the composite type and accumulates the type sizes of each member until m is reached. To clarify this procedure, let us consider a variable x of type A denoting a structure that entails the variables of figure 4.2: `struct A {int i; char c; char a[3];}`. The offset function then returns for $Offset(x, a)$ the following:

$$\begin{aligned} Offset(x, a) &= sizeof(i) + pad_0 + sizeof(c) + pad_1 \\ &= 4 + 0 + 1 + 0 \\ &= 5 \end{aligned}$$

with the assumption all paddings being zero. A field reference $x.a$ then leads to

$$\frac{\dfrac{\&x = SymLoc(x)}{\Sigma \models x \Downarrow_l \&x}\ \textsc{Var} \qquad F = \&x + Offset(x, a)}{\Sigma \models x.a \Downarrow_l \&x + 5}\ \textsc{Field}$$

The counterpart of the left-hand side evaluation constitutes the same of the right-hand sides. This leads us to the rules of $\Downarrow_v$ (see figure 4.4b). All of the expressions in the form of ϵ_ℓ are evaluated by dereferencing the corresponding location denoted by the rule Lhs. Besides, the value of a constant is its canonical representative, whereas the address-of operator is evaluated by returning the location of its operand. The more special case of a function call expression is handled by providing a symbolic variable $v \in Var'$ for its return value. The function $SymVal()$ exemplifies this behavior. The Cast rule enables us to cope with the premise of binary expressions in rule BinExp, that restricts all operands to have the same type. Further, we provide a separate rule to handle pointer arithmetic. In this special case of a (commutative) binary expression, the types of the operands are considered. Similar to the offset calculation concerning array subscripts, a multiple of the numeric operand's value is added or subtracted considering the size of the pointer's target. There is no rule for the case of both operands being pointers as its application might directly offend memory safety properties.

Invalid Dereference At this point, it is important to mention that whenever a memory access in the form of $\Sigma[E]^n$ causes an invalid dereference, the evaluation leads to the result *nil* as defined in 4.2.3. This behavior is crucial for the transfer relation, as the memory safety properties depend on whether a failed memory access occurred on a left- or right-hand side expression.

The previous section introduced the procedure of evaluating expressions to the associated locations, respectively values. Based on this, the next part of the thesis gives insights to the construction of SL-formulae.

4.2.5 Transfer Relation

The heart of our approach constitutes the transfer relation as part of the CPA. The transfer relation returns the successors for each outgoing edge of a state. Thereby, the CFA provides us different types of edges related to the statement the respective edge is labeled with. For all of them, the transfer relation implements the corresponding semantic rules that define the behavior of our analysis. We denote an abstract state as a symbolic heap $\Pi \wedge \Sigma$. In the following, we discuss each edge and provide the associated semantic rules.

Declaration Edge A variable declaration allocates memory on Σ according to the variable's type. The initialization of the allocated cells is not further specified, since the initial assignment can be modeled by an additional statement edge.

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

Statement Edge In C there are two essential groups of statements: assignments and "pure" statements. Concerning the former, the location of the left-hand side as well as the value of the right-hand side are determined. Afterwards, the heap is updated accordingly:

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

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

As the left- and right-hand side are independently processed of each other, the analysis is able to distinguish between an invalid dereference while reading from

($\perp^R$) and writing to ($\perp^W$) memory. Likewise, the pure statements are treated as they only differ in the absence of the left-hand side.

However, there are a few special cases that deserve a discussion in more detail. We cope with dynamic memory allocation using `malloc()` from `stlib.h` by evaluating the passed parameter ϵ defining the segment size. Afterwards, the given amount of memory cells is added to the spatial part Σ of our heap formula.

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

The function `alloca()` is analogously treated. In order to model the return value of `malloc()`, the function $SymVal()$ provides a symbolic variable x that is written to the left-hand side's location. By doing so, we assume on the one hand that the allocation by `malloc()` is always successful, on the other hand we require a deterministic value for κ, that can be statically evaluated. We provide an additional rule for a trivial case that directly leads to a memory leak, namely if the heap pointer returned by `malloc()` remains unutilized.

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

As the counterpart of dynamic allocation, we further provide rules for the deallocation using `free()`. In order to identify the associated segment of the heap pointer, we introduce the function $segmentSize(E)$. The function works as a lookup table including all memory segments with their associated size. If E is not a start address of such a segment -1 is returned (see 5).

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

At this point, it is crucial to establish the allocation check of address ϵ solely on the dynamic part of the heap Σ^{dyn} in order to model the function of `free()` correctly. The event that there is no match for the value of ϵ leads to an invalid free $\perp^F$ violating the memory safety properties.

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

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

Besides, whenever memory is released the resulting state has to be checked for potential leaks. We further discuss this memory safety property together with variable scope in the last paragraph of this section.

Assumption Edge The CFA provides us with assumption edges in order to model control flow paradigms like conditionals or loops. Each of them comes with an assumption expression ϵ that functions as a constraint for the potential successor(s). We evaluate ϵ according to Σ and conjugate it to the pure formula part of the successor's state. Finally, the pure formula $\Pi \wedge C$ is checked whether it is satisfied or not. Concerning the latter, there is no successor to be reached and the termination is ensured. For each case, the corresponding rule is provided.

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

$$\frac{\Sigma \models \epsilon \Downarrow_v C \quad \Pi \wedge C \text{ is not SAT}}{\{\Pi \wedge \Sigma\}\ \epsilon\ \{\top\}}\ \textsc{Assume}^{-}$$

Function Call Edge A function that is defined inside the executed program itself is modeled by the function call edge. The edge describes the function entry node. Here, its parameters and space for a potential return value are allocated according to the function type. In this respect, $SymLoc()$ provides us the corresponding symbolic locations. We write $SymLoc(f)$ to address the location reserved for the return value and $SymLoc(f_n)$ for the n-th parameter, respectively. Therewith, we formalize the memory segment of a `void` function f with n parameters as

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

with F_i being the value of each parameter: $\Sigma \models \epsilon_i \Downarrow_v F_i$. In case of a function with a return value, $\Sigma^R()$ defines its reserved memory:

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

Taken together we are able to define the semantic rules for a function call edge.

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

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

The rules differ merely in the function's return type. If a return value is present, its associated memory segment is conjugated using the spatial conjunction illustrated by the second rule FunCall. Further, each parameter expression might cause an invalid memory access:

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

Return Statement Edge The return statement edge models the assignment of the computed return value to the previously allocated return variable space. In case that the function f has no return value nothing has to be done.

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

Otherwise, the passed expression has to be evaluated and assigned to the corresponding location.

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

Function Return Edge As a counterpart to the function entry node, the function return edge models the exit node. Here, the previously allocated memory space for the parameters and the return value has to be released. We define the following rules coping with the described behavior:

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

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

As a special case, all memory that was allocated due to `alloca()` has to be disposed as well. We give further insights to this procedure in the scope of chapter 5.

Variable Scope For each edge, the CFA provides a set of variables, which go out of scope for the next successor. Our analysis makes use of this feature to accomplish an efficient way to determine memory leakage: Only the part of memory that was addressed by the released variables has to be examined. Thereby, a memory leak occurs when the only pointer to a memory cell allocated

on Σ^h is lost. We formalize this characteristic by defining a function reachable

$$\begin{aligned} reachable(\Sigma^s * \Sigma^h, E) := \ & \Sigma^s \models Allocated(E) \\ & \vee (\exists F, G.(F \mapsto G \wedge E = G \wedge reachable(\Sigma^s * \Sigma^h, F))) \end{aligned}$$

saying that a memory cell at address E is reachable, if it is either allocated on the non-dynamic part of the heap Σ^s, or if there exists an alias G. In case of the latter, the cell at address F containing the value G is recursively checked for reachability.

As the above definition might lead to endless recursion in case of a cyclic list where chunks of memory are referencing each other, one has to remember those locations that have already been processed. For the sake of completeness, an alternative formalization is stated in the appendix that copes with this criterion (see A.1).

If now a variable goes out of scope and thus has to be removed from memory, only the effected heap cells are analyzed according to potential memory leaks. Thus, we define the function $Leak()$ dependent on the value of the variable that went out of scope.

$$Leak(\Sigma^s * \Sigma^h, E) := \Sigma^h \models Allocated(E) \wedge \neg reachable(\Sigma^s * \Sigma^h, E)$$

Therewith, a memory leak occurs when the value E is the address of a memory cell in Σ^h that is furthermore not reachable anymore. To bring both formulae together, we provide semantic rules that are applied for any edge with variables that go out of scope for the successors.

$$\frac{\begin{array}{ll} \Sigma \models x \Downarrow_l E & n = sizeof(x) \\ \Sigma \models x \Downarrow_v F & \bigwedge_{i=0}^{n-1} Leak(\Sigma, F^i) = false \end{array}}{\{\Pi \wedge \Sigma * E \mapsto F^0, ...F^{n-1}\}\ oos(x)\ \{\Pi \wedge \Sigma\}}\ \textsc{OutOfScope}$$

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

We have introduced the operational semantics that cope with the different types of edges provided by the CFA. In order to achieve a pointer analysis, the discussed transfer relation and abstract domain have to be implemented as part of an independent CPA. We illuminate this procedure in the next chapter.

5 Implementation

The theory part of this thesis abstracts from several details in order to bring out the most important aspects of the approach. However, a concrete implementation can only be achieved by a precise specification. In this spirit, we now give insights to the used technologies and further discuss the construction of the SL-CPA.

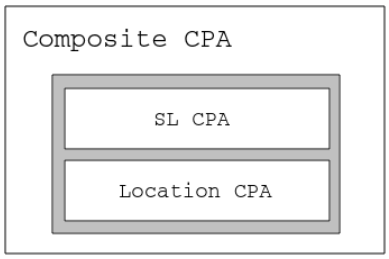


Figure 5.1: *CPA structure*

The whole implementation is written in Java and constitutes a distinct CPA implementation of the CPAchecker framework. Thereby, the actual pointer analysis is a composition of our SL-CPA and the Location CPA (see figure 5.1). The latter provides the meaningful transfer edges $e \overset{g}{\rightsquigarrow} e'$ through a syntactic reachability analysis for all control-flow edges of the CFA. The operational semantics are then applied by the SL transfer relation for each transfer given by the Location CPA.

5.1 Symbolic Heap Formula

Figure 4.1 has introduced the grammar of symbolic heap formulae. With the help of JavaSMT, we construct a formula of this grammar that suffices the SMT-LIB standard format. This further requires each formula to be strongly typed in respect to locations and values. Since we designed the memory model to be of the precision of bytes, bit-vectors are used for both. Thereby, locations

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

Figure 5.2: *The SL state*

are bit-vectors of length eight times `sizeof(*void)`, respectively `sizeof(char)` for values. At this point, we refer to the functions $SymLoc()$ and $SymVal()$ introduced in the scope of the operational semantics of 4.2.4. Symbolic locations or values returned by $SymLoc()$ respectively $SymVal()$ are bit-vector variables with unique names including the function scope and SSA-index.

5.2 SL State

Each element of the abstract domain is represented by a SL state shown in figure 5.2. The SL state encapsulates two maps of type `LinkedHashMap` which represent the spatial part of the symbolic heap $\Sigma^s * \Sigma^h$. The persistent insertion order of the maps enables us to resolve consecutive memory segments more efficiently. The pure part of a formula represents the conjugation of constraints as `BooleanFormula`. In contrast to the spatial part, Π is already a formula because a predicate constraint once conjugated to Π will never be removed for a successor.

Besides, a SL state tracks the size of all memory segments. In this respect, the SL state provides a map *SegmentSizes* between the start address of a segment and its associated size. The map is updated whenever a segment is (de)allocated due to a variable declaration or a dynamic memory allocation. With this, we are able to dispose allocations caused by `free()`. The function $SegmentSizes()$ stands in direct relation to this implementation detail. Further, as the map only contains the start address of each segment, invalid frees are also detected.

In terms of allocation, the analysis also copes with the function `alloca`. In this special case, the allocated memory is only available in the function scope where `alloca` was called. Therefor, the SL state includes *Allocas*, which maps a function to the start addresses of all memory segments allocated by `alloca`.

Since all memory - according to the C standard - is automatically released when the function scope is left, our transfer relation implements the same as part of the function return edge.

We previously assumed, that a program has exactly one declaration for each variable in order to achieve static typing. However, there are cases of local declarations (cf. example program 1.1a, line 4) that make a more specific treatment necessary. For this reason, the SL state has a `SSAMap`, which is updated whenever a variable is declared or assigned. Respectively, each symbolic location returned by $SymLoc()$ is clearly related to a variable.

Last but not least, the SL state includes the memory safety properties that have been offended by the control flow edge between the state and its predecessor. In this fashion, the set $Properties$ is utilized to report the result of the analysis.

5.3 Solver

The constructed formulae eventually have to be analyzed by a solver. In this concern, JavaSMT provides with CVC4 [19] a SMT solver that is additionally capable of SL theory without quantifiers. Furthermore, CVC4 supports bit-vectors and has also participated in the SL-COMP [21].

Allocation Check Whenever a chunk of memory is accessed, the corresponding location has to be checked for allocation. As the term $\Sigma \models Allocated(E)$ implies a semantic check for E, a syntactical lookup alone is not sufficient. Therefor, we need CVC4 in order to prove the allocation of E.

In this concern, the spatial part of the associated state is converted to a SL-formula by the recipe of algorithm 3. For reasons of efficiency, first a trivial

Algorithm 3: Allocation check in terms of $\Sigma \models Allocated(E)$

Input: a symbolic location E as `BVFormula` and a `Map` M describing a symbolic heap
Output: *true* if allocated, $false$ otherwise

if *M.containsKey(E)* **then**
  return *true*;
end
$\Sigma = emp$;
foreach *(key, value)* $\in M$ **do**
  $\Sigma = \Sigma * (key \mapsto value)$;
end
return $\neg SAT(\Sigma * E \mapsto -)$;

syntactic check is performed. If no match is found, the map is converted to a symbolic heap formula. Points-to predicates for all key-value pairs of the map are created and conjugated with each other using the spatial conjunction $*$. We point out, that it is not always necessary to consider $\Sigma^s * \Sigma^h$ as a whole, since for instance variables do not occur on Σ^h, whereas memory segments that shall be freed must not to be found on Σ^s either. After the formula is constructed, another points-to predicate with the location E pointing to some free variable is conjugated. The resulting symbolic heap is then passed to the solver. By the definition of $*$, the satisfiability of the formula implies that E cannot be allocated yet. Hence, the negation of the solver call result is returned. However, a drawback of this implementation is that the *antiframe* is not explicitly determined. In the case of E being an alias, meaning a *semantic* but not a syntactic equivalent to this particular cell, one cannot retrieve the matching map entry.

The function $Deref()$ - defined in section 4.2.3 - targets this issue. In respect to that, we now provide the corresponding algorithm 4 as part of our implementation. The algorithm is slightly different to the former one as it performs a solver call for each memory cell in order to determine the allocated cell. In this way, it is possible to retrieve the antiframe as each heaplet is independently checked whether it leads to an inconsistent symbolic heap or not. Nevertheless, this procedure leads to a drastic increase of solver calls. Further, as the SL-formula of $key \mapsto value * E \mapsto -$ is semantically the same compared to a trivial equivalence check $key = E$, another algorithm is presented.

In order to perform an equivalence check over a set of locations, only a SMT solver is necessary. As algorithm 5 illustrates, even multiple solver calls can be

Algorithm 4: Dereferencing a memory location $Deref(\Sigma, E)$

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
foreach (key, value) ∈ M do
    if ¬SAT(key ↦ value * E ↦ −) then
        return (key, value);
    end
end
return nil;
```

Algorithm 5: Dereferencing with SMT solver $Deref^{ModelSAT}(\Sigma, E)$

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 = ⋀_{k∈M.keys()} k = E ⟺ aux_k;
if SAT(formula) then
    model = getModel(formula);
    foreach (aux_k, value) ∈ model do
        if value then
            return k;
        end
    end
end
return nil;
```

avoided by introducing boolean auxiliary variables each related to an unique key. Afterwards, a model is generated and the key which is linked to the auxiliary variable that evaluated to *true* is returned.

Once the location is determined, the corresponding value is computed regarding the segment size used in algorithm 6. Here, the persistent insertion order of the entries provided by the `LinkedHashMap` is crucial for the correctness of the algorithm. Therewith, expensive solver calls are saved under the premise that consecutive memory cells of the same segment are inserted in the map accordingly.

Feasibility Check In the case of an assumption edge, the transfer relation provides a successor, only if the corresponding assumption leads to a satisfiable pure part of the symbolic heap (see 4.2.5). Though, this requires another solver call in order to ensure the termination of the analysis.

In this chapter, we worked out how the abstract domain and transfer relation are embedded into a concrete CPA implementation of the CPAchecker framework. Thereby, the allocation check is emphasized to be the crucial part of the analysis. The algorithms 4 and 5 introduced procedures which cope with these allocation checks by additionally providing the values of the cells. However, the

stated algorithms either lead to an increased amount of solver calls or a potentially complex model generation. Both might lead to significant performance limitations. We want to illuminate this conjecture in the evaluation part of this thesis.

Algorithm 6: Dereferencing a memory segment $\Sigma[E]^n$

Input: a symbolic location E as `BVFormula`, a `LinkedHashMap` M describing a symbolic heap and a segment size n
Output: a `BVFormula` denoting the value of the segment starting at address E if allocated, nil otherwise

```
// Get start address of the segment.
(loc, val) := Deref(M, E);
if (loc, val) == nil then
    return nil;
end
// Construct value.
found := false;
i = n;
res = nil;
foreach (key, value) ∈ M do
    if i==0 then
        break;
    end
    found = found ∨ key == loc;
    if found then
        res = res :: value; // Little-endian
        i = i − 1;
    end
end
return res;
```

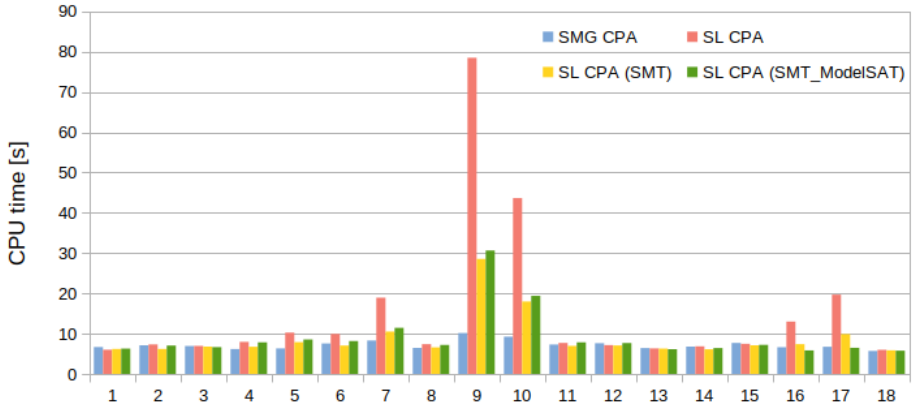
6 Evaluation

In order to achieve reasonable information, we compared the analysis to an already established procedure based on SMGs. The corresponding implementation as part of the CPAchecker framework constitutes the SMG-CPA. We evaluated our approach using a small benchmark set `memsafety-ext3`. It is part of the SV-Benchmarks[1] collection in the version as included in the SV-COMP 2020 [3]. The benchmark set comprises 18 distinct tasks all of them matching the category *MemorySafety* of the SV-COMP 2020. The following features of C were covered among others: pointer arithmetic with aliasing; dynamic allocation using `malloc()`, `realloc()` and `alloca()`; function calls and variable scope.

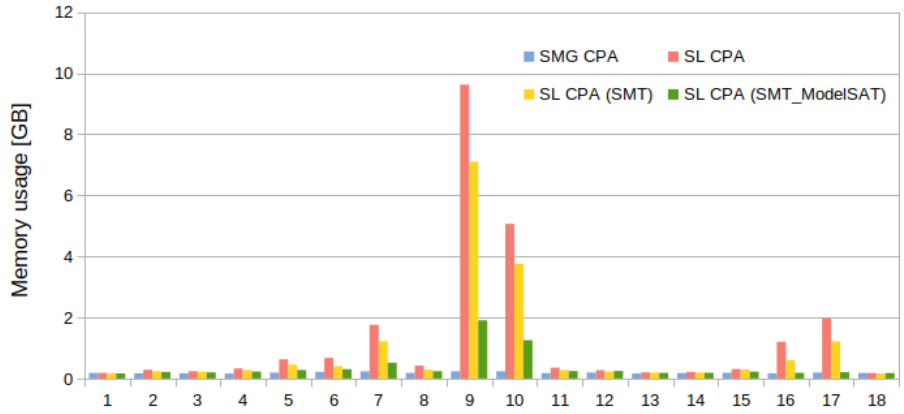
6.1 Execution Environment

The tasks were executed on machines with Intel Xeon E3-1230 v5 CPUs,3.40 GHz CPU frequency, and 33 GB RAM. The execution time was limited to 90 seconds with the use of 15GB RAM and two CPU cores. We utilized BenchExec [6] to measure the consumed memory and the elapsed time. Further, the number of solver calls and the overall solver call time was determined. Thereby, all solver calls were executed using CVC4 and the discussed data is solely based on successfully terminated tasks with the correct property result. Two exceptional cases denote tasks 16 and 17 that lead to segmentation faults caused by a solver call related to algorithm SMT_ModelSAT. Besides, we renounced the calculation of mean values of multiple runs because of clear differences between all algorithms. The experiments refer to the CPA implementations of the `sl-integration0` branch as part of the CPAchecker subversion repository[2]. The revision is `r34851` and any of the used configuration files according to BenchExec and the CPA can be found in the appendix (see A.2 and A.3).

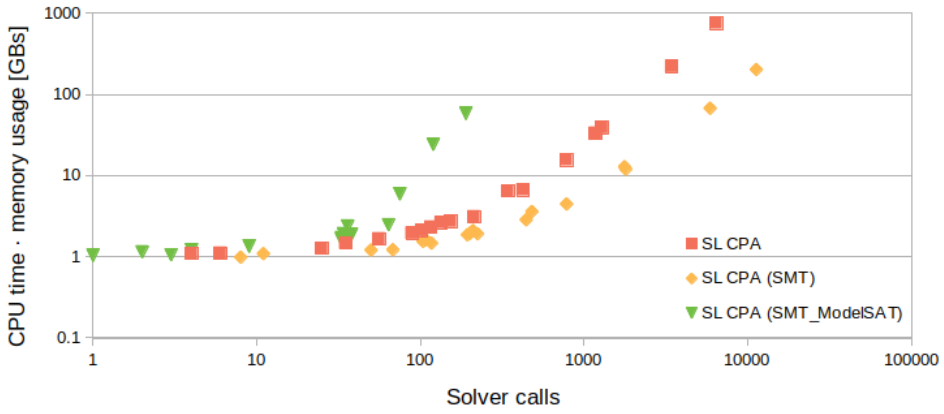
[1] `https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20/c/memsafety-ext3`
[2] `https://svn.sosy-lab.org/software/cpachecker/branches/sl-integration0/`



(a) Elapsed CPU time per task



(b) Used memory per task



(c) Comparison of SLCPA implementations

Figure 6.1: *The elapsed CPU time (6.1a) and used memory (6.1b) as well as the combination of both in 6.1c*

6.2 Results

The figures of 6.1a and 6.1b illustrate the performance of the SMG-CPA compared to the different algorithms of the SL-CPA. For each algorithm discussed in 5 we provide an evaluation with the setup described above: the red pillars (SL) refer to algorithm 4 based on SL; yellow refers to the same algorithm except that trivial equivalence checks are performed (SMT); the green pillars represent algorithm 5 as an optimization of the latter by reducing the amount of solver calls (SMT_ModelSAT).

As one can see, the SMG-CPA has a low volatility across the entire test set, whereas the SL-CPA has a CPU time and memory usage increase in a couple of tasks across all algorithms. Especially tasks 9 and 10 are more time- and memory-consuming. Although the SMT-based algorithms are slower and less memory efficient than the SMG approach, they still are significantly better than the SL algorithm. Besides, the optimization by SMT_ModelSAT is reflected in the associated memory consumption, which is much lower compared to the other two SL-CPA implementations.

Figure 6.1c further compares the SL-CPA algorithms by plotting the amount of solver calls on the x-axis and the efficiency - modeled as the product of consumed memory and elapsed CPU time - on the y-axis. Again, the advantage of the SMT_ModelSAT algorithm is illustrated. But also SMT performs better than SL, even though more solver calls are executed. Although one might challenge the equal weighting of memory and time consumption, the example nonetheless retains its meaning.

According to the time elapsed during solver calls, figure 6.2 gives more insights. It clarifies that the higher execution time of the SL algorithm is mainly caused by solver calculations. Furthermore, the decrease of solver calls related to the SMT_ModelSAT algorithm does not lead to a significant increase of execution time, since it is compensated by a reduced Δ-time.

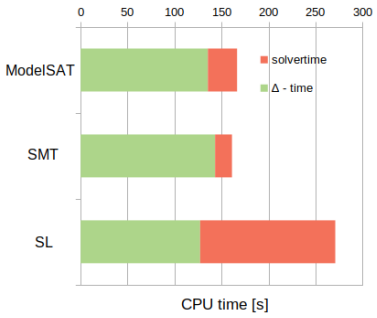


Figure 6.2: *Assignment of CPU time to solvertime (red) and other computations (green)*

The evaluation has shown the advantages and drawbacks of the different algorithms. It clearly exposed the lack of performance of SL for alienated procedures such as equivalence checks that SMT solvers are much more suitable for. This strongly motivates the extension of the SL solver interface oriented towards problems such as allocation checks and dereferencing.

7 Discussion

On the one hand, the evaluation has demonstrated the successful application of our approach in practice. Though, on the other hand, it has simultaneously revealed its limitations. In this concern, we want to highlight the gained knowledge and clarify the important aspects that have emerged in the scope of this thesis.

We applied a pointer analysis based on symbolic execution in order to track a program's state in terms of memory manipulation. Therewith, we successfully implemented a CPA into the CPAchecker framework that uses a SL solver interface provided by JavaSMT.

A fundamental part of the approach constitutes the memory model. The abstraction level of symbolic locations mapping to symbolic values allows us to represent the side effects of memory-involved operations. The use of bit-vectors for all parts of the model enables a generic application. At the same time, composite types such as structures and also more complex data structures including lists are supported. Furthermore, the model can either be directly transferred into SL, or alternatively, particular parts can be analyzed by SMT solvers without the awareness of a spatial context. In this respect, the combination of SMT and SL has proven to yield high potential (e.g. Assumption Edge).

However, there are several issues that have been encountered by our approach. These and the corresponding motivations are stated in the following:

Quantification and Abduction The main issue denotes the missing tool addressing allocation check and dereferencing. In this concern, the resolution of symbolic locations has to be improved by the means of a suitable solver interface. This can be achieved by implementing a SL theorem prover that is aware of quantified symbolic heaps and abduction techniques in order to provide results for F and G according to an extended allocation check

$$\Sigma \models \mathit{Allocated}(E) \iff \Sigma \models \exists \Sigma', F, G.(E = F \land \Sigma = \Sigma' * F \mapsto G).$$

Furthermore, the support of quantification allows a treatment of nondeterministic values at compile time. This is of substantial importance for dynamic memory allocation or external function declarations among others.

Scalability Another important aspect denotes the scalability. Thereby, one of the key strength of SL is the spatial division of memory provided by the frame rule:

$$\frac{\{P\}C\{Q\}}{\{P * F\}C\{Q * F\}}$$

By observing only the part of memory that is affected by a program, an analysis can be subdivided into smaller tasks. Nevertheless, this procedures requires the frame inference of F which is currently not supported by the used solver interface.

Segment Predicate Our approach abstracts the memory as a collection of byte-sequences. Thereby, we conjugate multiple heaplets forming a consecutive data segment according to the segment size. In this spirit, an additional SL predicate targeting this procedure would simplify the treatment of types and further increase readability.

CPA Composition The CPAchecker framework allows a composition of multiple CPAs for the same analysis. By this, features of the analysis can be distributed. One example targeting this consideration constitutes the feasibility check for assumption edges. Currently, a potentially expensive solver call is used to check whether a pure formula is satisfiable or not. With the use of a `ValueAnalysisCPA`, such a call might not be necessary in most cases and thus further performance increase can be achieved.

This thesis has demonstrated that SL is indeed a powerful theory in order to assess the behavior of programs according to memory. Especially in terms of scalability, SL has high potential in the area of static program analysis. And combined with a SMT solver the performance can be increased once more. Nevertheless, its practical application depends a lot on the features provided by the underlying solver interface.

Appendix A

A.1 Reachability check with cycles

$$
\begin{aligned}
reachable(\Sigma^{+} * \Sigma^{dyn}, R, E) := \Sigma^{+} \models Allocated(E) \\
\vee (\nexists F \in R.(E = F) \\
\wedge \exists F, G.(F \mapsto G \wedge E = G \\
\wedge reachable(\Sigma^{+} * \Sigma^{dyn}, R \cup \{G\}, F)))
\end{aligned}
$$

$$
Leak(\Sigma, E) := \neg reachable(\Sigma, \{\}, E)
$$

A.2 SLCPA Configuration - sl.properties

```
# Separation Logic CPA
cpa = cpa.arg.ARGCPA
ARGCPA.cpa = cpa.composite.CompositeCPA
CompositeCPA.cpas = cpa.location.LocationCPA, cpa.sl.SLCPA
solver.solver = CVC4
specification = specification/memorysafety.spc
memorysafety.config = sl.properties

# SL
cpa.sl.allocationCheckProcedure = SL

# PathFormulaManager
cpa.predicate.ignoreIrrelevantVariables = false
cpa.predicate.handlePointerAliasing = false
cpa.predicate.handleSL = true
```

A.3 BenchExec - integration-sl.xml

```
<?xml version="1.0"?>
<!DOCTYPE benchmark PUBLIC
"+//IDN sosy-lab.org//DTD BenchExec benchmark 1.0//EN"
"http://www.sosy-lab.org/benchexec/benchmark-1.0.dtd">
<benchmark tool="cpachecker"
  timelimit="90s"
  hardtimelimit="120 s"
  memlimit="15 GB"
  cpuCores="2">
  <rundefinition/>
  <option name="-stats"/>
  <tasks name="MemSafety-ext3">
    <includesfile>
      ../programs/benchmarks/MemSafety-SL.set
    </includesfile>
    <propertyfile>
      ../programs/benchmarks/properties/valid-memsafety.prp
    </propertyfile>
    <option name="-sl"/>
    <!-- One of the following procedures: SL, SMT, SMT_MODELSAT -->
    <option name="-setprop">cpa.sl.allocationCheckProcedure=SL</option>
  </tasks>
  <columns>
    <column title="solvertime">solvertime</column>
    <column title="solvercalls">solvercalls</column>
  </columns>
</benchmark>
```

Bibliography

[1] C. Barrett, M. Deters, L. de Moura, A. Oliveras, and A. Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013. URL http://doi.org/10.1007/s10817-012-9246-5.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*, pages 52–68. Springer, Berlin, Heidelberg, 2005. ISBN 978-3-540-32247-4. URL https://doi.org/10.1007/11575467_5.

[3] D. Beyer. SV-Benchmarks: Benchmark set of 9th Intl. Competition on Software Verification (SV-COMP 2020). Zenodo, 2020. URL https://doi.org/10.5281/zenodo.3630188.

[4] D. Beyer and M. E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011. URL https://doi.org/10.1007/978-3-642-22110-1_16.

[5] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007. ISBN 978-3-540-73367-6. URL https://doi.org/10.1007/978-3-540-73368-3_51.

[6] D. Beyer, S. Löwe, and P. Wendler. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer*, 21(1):1–29, 2019. URL https://www.doi.org/10.1007/s10009-017-0469-y.

[7] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint Analysis: A Shape Analysis That Discovers Preconditions", booktitle="Static Analysis. pages 402–418. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74061-2. URL https://doi.org/10.1007/978-3-540-74061-2_25.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Space Invading Systems Code. In *Logic-Based Program Synthesis and Transformation*, pages 1–3, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00515-2. URL https://doi.org/10.1007/978-3-642-00515-2_1.

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

[10] Cristiano Calcagno, Dino Distefano, and Peter W. O'Hearn. Open-sourcing Facebook Infer: identify bugs before you ship, 2015. URL https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/. Online; accessed August 31, 2020.

[11] 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. URL https://doi.org/10.1007/11691372_19.

[12] K. Dudka, P. Peringer, and T.áš Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Proc. CAV*, pages 372–378. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22110-1. URL https://doi.org/10.1007/978-3-642-22110-1_29.

[13] K. Dudka, P. Peringer, and T.áš Vojnar. Byte-Precise Verification of Low-Level List Manipulation. In *Static Analysis*, pages 215–237. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38856-9. URL https://doi.org/10.1007/978-3-642-38856-9_13.

[14] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969. URL https://doi.org/10.1145/363235.363259.

[15] 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. URL https://doi.org/10.1007/978-3-642-54862-8_28.

[16] 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. URL https://doi.org/10.1007/3-540-44802-0_1.

[17] C. Peirce. *Collected papers of Charles Sanders Peirce*. Harvard University Press, 1960. ISBN 0674138023, 9780674138025. URL https://books.google.de/books?id=G7IzSoUFx1YC.

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

[19] A. Reynolds, R. Iosif, C. Serban, and T. King. A Decision Procedure for Separation Logic in SMT", booktitle="Automated Technology for Verification and Analysis. pages 244–261. Springer International Publishing, 2016. ISBN 978-3-319-46520-3. URL https://doi.org/10.1007/978-3-319-46520-3_16.

[20] J. Reynolds. Separation logic: A logic for shared mutable data structures. pages 55– 74, 02 2002. ISBN 0-7695-1483-9. URL https://doi.org/10.1109/LICS.2002.1029817.

[21] 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. URL https://doi.org/10.1007/978-3-030-17502-3_8.

[22] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A Practical Verification Framework for Preemptive OS Kernels. In *Proc. CAV*, pages 59–79. Springer International Publishing, 2016. ISBN 978-3-319-41540-6. URL https://doi.org/10.1007/978-3-319-41540-6_4.