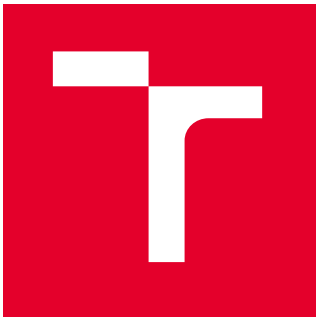


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS OF HEAP-MANIPULATING PROGRAMS USING SEPARATION LOGIC

STATICKÁ ANALÝZA PROGRAMŮ PRACUJÍCÍCH S DYNAMICKOU PAMĚTÍ S VYUŽITÍM SEPARAČNÍ LOGIKY

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR — Tomáš Brablec

AUTOR PRÁCE

SUPERVISOR — Ing. Tomáš Dacík

VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

163339

Institut:	Department of Intelligent Systems (DITS)
Student:	**Brablec Tomáš**
Programme:	Information Technology
Title:	**Static Analysis of Heap-Manipulating programs using Separation Logic**
Category:	Formal Verification
Academic year:	2024/25

Assignment:

1. Study methods for static analysis of heap-manipulating programs based on separation logic.
2. Study the platform Frama-C and the Astral solver for separation logic.
3. Desing an analyser of the Frama-C platform using Astral with focus on programs with linked lists and their variants.
4. Implement the proposed approach.
5. Evaluate the analyser on suitable programs with focus on the analysis precision and time spend by solving formulae of separation logic.
6. Summarise achieved results and discuss possible directions for future work.

Literature:

- Dino Distefano, Peter W. O'Hearn, Hongseok Yang. A Local Shape Analysis Based on Separation Logic. TACAS 2006.
- Dacík Tomáš, Rogalewicz Adam, Vojnar Tomáš and Zuleger Florian. Deciding Boolean Separation Logic via Small Models. TACAS 2024.
- Berdine, J. *et al.* (2007). Shape Analysis for Composite Data Structures. CAV 2007.
- Nikolai Kosmatov, Virgile Prevosto, Julien Signoles. Guide to Software Verification with Frama-C. ISBN 978-3-031-55610-4.

Requirements for the semestral defence:
The first three points of the assignment and at least the beginning of work on the fourth point.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Dacík Tomáš, Ing.**
Consultant:	Vojnar Tomáš, prof. Ing., Ph.D.
Head of Department:	Kočí Radek, Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.10.2024

Abstract

This thesis introduces the KTSN static analyzer that aims to verify the memory safety of C programs. The method focuses on programs that manipulate linked lists. Our tool is able to prove the correctness of handling dynamically allocated memory and detect invalid pointer dereferences such as use-after-free, memory leaks, and other bugs. The approach is based on dataflow analysis, and it uses separation logic to represent abstract memory states. The analysis uses Astral, a dedicated solver for SL based on translation to SMT. KTSN is implemented as a plugin to the Frama-C framework. The tool was tested on the linked lists subset of the SV-COMP benchmarks and compared to other verification tools. While it does not reach the top competitor, PredatorHP, it outperforms most verifiers in this category, including EVA, a value analyzer of Frama-C. Thus, our method shows promise of integration into EVA, greatly improving the analysis of linked lists in Frama-C.

Abstrakt

Tato práce představuje statický analyzátor KTSN zaměřený na verifikaci bezpečnosti práce s pamětí programů v jazyce C. Naše metoda se soustředí na programy, které pracují s lineárními seznamy. Náš analyzátor je schopen verifikovat správnost manipulace s dynamicky alokovanou pamětí a detekovat neplatné dereference ukazatelů (např. use-after-free), úniky paměti a další chyby. Náš přístup je založen na analýze toku dat a využívá separační logiku pro reprezentaci abstraktních stavů paměti. Nástroj je implementovaný ve frameworku Frama-C. Analýza využívá dedikovaný solver Astral pro rozhodování separační logiky založený na překladu do SMT. Nástroj KTSN byl otestován na podmnožině SV-COMP benchmarků zaměřené na lineární seznamy a porovnán s dalšími verifikačními nástroji ve své kategorii. Přestože nedosahuje úrovně nejlepšího nástroje PredatorHP, překonává většinu verifikačních nástrojů v této kategorii, včetně analyzátoru EVA, který je součástí frameworku Frama-C. Naše metoda rovněž vykazuje potenciál pro integraci do analyzátoru EVA, čímž by se výrazně vylepšila analýza lineárních seznamů ve Frama-C.

Keywords

verification, static analysis, separation logic, Astral, Frama-C, SV-COMP, KTSN

Klíčová slova

verifikace, statická analýza, separační logika, Astral, Frama-C, SV-COMP, KTSN

Reference

BRABLEC, Tomáš. *Static Analysis of Heap-Manipulating Programs using Separation Logic.* Bachelor's Thesis. Tomáš DACÍK (supervisor). Brno: Brno University of Technology, Faculty of Information Technology, 2025.

Rozšířený abstrakt

Existuje několik způsobů správy paměti v programovacích jazycích. Tyto způsoby se dělí na automatickou správu paměti, která využívá garbage collector k uvolňování již nedosažitelných alokací, a manuální správu paměti, kde musí kód programu explicitně uvolňovat alokace. Manuální správa paměti s sebou nese riziko několika typů chyb, kterých se může programátor při psaní kódu dopustit. Jednou z těchto chyb je tzv. use-after-free, neboli přístup do již uvolněné alokace. Dále jde např. o double-free – uvolnění již uvolněné alokace, nebo únik paměti, kdy dojde ke ztrátě všech ukazatelů na alokaci bez jejího uvolnění. Některé z těchto chyb mohou vytvořit bezpečnostní zranitelnosti v softwaru, zneužitelné např. ke spuštění kódu útočníkem.

K nalezení těchto chyb se používá řada technik. Jedním přístupem je testování a dynamická analýza programu, např. pomocí instrumentace kódu překladačem. Dalším je statická analýza programu a formální verifikace. Formální verifikace umožňuje ověřit korektnost programu, přičemž se typicky zaměřuje na ověření nějaké konkrétní vlastnosti. Z hlediska paměťové bezpečnosti nás zajímá právě korektnost všech přístupů do paměti.

Jednou z metod statické analýzy je analýza toku dat (dataflow), která spočívá v procházení grafu toku řízení (CFG) programu a postupné aktualizaci informací o programu asociovaných s každým uzlem v CFG. V kontextu analýzy práce s pamětí jsou pro nás relevantní hodnoty ukazatelů a datové struktury v dynamické paměti. K jejich reprezentaci se v těchto analýzách často využívají formule separační logiky, která nám umožňuje efektivně reprezentovat datové struktury o neomezené velikosti.

Tato práce popisuje tvorbu statického analyzátoru KTSN pro programy v jazyce C, který bude schopen verifikovat korektnost práce s dynamicky alokovanou pamětí se zaměřením na lineární seznamy. K tomu je využita metoda analýzy toku dat, a stavy programu jsou reprezentovány pomocí formulí separační logiky. K implementaci analýzy je využit framework Frama-C, který poskytuje zjednodušenou formu AST analyzovaného programu, rozhraní pro transformace tohoto AST a framework pro implementaci dataflow analýzy. K vyhodnocování splnitelnosti formulí separační logiky je použit solver Astral. Ověření splnitelnosti formulí je třeba k nalezení abstraktního stavu paměti popisujícího cykly (tzv. fixpoint), což je nutné k ukončení analýzy.

Samotná implementace se skládá z několika částí. Prvně je to předzpracování analyzovaného AST, které slouží několika účelům. Jedním účelem je zjednodušit komplexní výrazy v příkazech přiřazení, argumentech funkcí a podmínkách. Dalším účelem je analyzovat struktury definované v programu a určit, který typ seznamu popisují. Toto se provádí na základě heuristiky, která odvodí typ seznamu podle typu položek struktury.

Další částí implementace je analýza jednotlivých příkazů. Při dosažení příkazu je třeba upravit formule reprezentující stav programu před vykonáním příkazu tak, aby popisovaly stav po vykonání příkazu. Tyto příkazy zahrnují několik druhů dereferencí a přístupů na položky struktury, alokace a uvolňování paměti, a volání funkcí.

Implementace dále obsahuje zjednodušování a abstrakce formulí na přechodu mezi dvěma příkazy. Zjednodušování formulí slouží primárně ke zkrácení času, který potřebuje solver k ověřování jejich splnitelnosti. Do této kategorie spadá například odstraňování nedosažitelných prostorových predikátů z formulí (tyto jsou pak hlášeny jako úniky paměti), odstraňování nepotřebných proměnných z ekvivalencí, nebo přejmenování proměnných, kterým v programu skončil rozsah platnosti. Abstrakce spočívá v nalezení ukazatelových predikátů ve formuli tvořících řetězec a jejich spojení do seznamových predikátů, které popisují lineární seznam o neomezené délce. Tento krok je nutný k nalezení fixpointu pro cykly.

Analyzátor byl otestován na ručně vytvořené sadě testovacích programů, které pracují se všemi podporovanými typy seznamů. Do testovaných operací patří alokace a uvolnění seznamu o nedeterministické délce, iterace skrz prvky seznamu, přidání a odstranění prvků ze seznamu nebo obrácení pořadí seznamu. Analyzátor byl schopen ověřit korektnost všech těchto jednoduchých programů. Při přidání chyb do těchto programů byl nástroj schopný tyto chyby detekovat. Mezi detekované chyby se řadí dereference ukazatelů na dealokovanou paměť, dereference ukazatele, který může mít nulovou hodnotu, únik paměti, nebo dealokace již uvolněné paměti.

Další testování proběhlo na datasetu ze soutěže SV-COMP, konkrétně na podmnožině programů pracujících s lineárními seznamy. Tento dataset obsahuje celkem 134 programů testujících velké množství operací na mnoha typech seznamů. Do těchto programů patří např. řazení seznamů, průchody a modifikace cyklických seznamů nebo kombinace několika typů seznamů v jednom programu. Na tomto datasetu je KTSN schopný správně analyzovat 80 programů, z toho 74 jsou ověření korektních programů a zbylých 6 jsou úspěšné detekce chyb. Tento výsledek překonává všechny analyzátory v dané kategorii kromě dvou (PredatorHP se 124 správnými výsledky a CPAchecker se 118 správnými výsledky). Analyzátor EVA, postavený nad frameworkem Frama-C, dokáže správně analyzovat 56 programů.

Současná implementace analýzy má několik omezení, která znemožňují dosáhnout lepších výsledků. Prvně je to chybějící podpora analýzy ukazatelové aritmetiky. V současném stavu není možné obecně reprezentovat číselný offset do struktury ve formulích separační logiky. Toto znemožňuje analyzovat 25 testů z SV-COMP datasetu, které používají ukazatelovou aritmetiku k přístupu do uzlů seznamu. Tyto testy selžou už při předzpracování kódu.

Dalším omezením je absence analýzy číselných hodnot v programu, což znemožňuje přesnější analýzu podmínek v cyklech. Ta je nutná například k přesnější detekci chyb v programech. Neschopnost analyzovat jiné než ukazatelové podmínky způsobí, že 18 SV-COMP testů skončí neznámým výsledkem, jelikož se nepodaří prokázat přítomnost chyby.

Zbylých 11 selhávajících testů skončí překročením časového limitu pro analýzu, například kvůli tomu, že implementují nepodporovaný typ seznamu, nebo protože heuristika pro odvození typu seznamu vrátila špatný výsledek.

Nabízí se několik možných směrů budoucího rozšíření analyzátoru. Jednou možností je přidat jednoduchou analýzu číselných hodnot přímo do nástroje KTSN. Separační logika v podobě, v jaké ji implementuje solver Astral, umožňuje vložit do formulí termy v SMT, které mohou reprezentovat hodnoty číselných proměnných. Zajímavějším směrem vývoje je integrace naší metody přímo do analyzátoru EVA. To by nám umožnilo přímo v průběhu analýzy získávat informace o číselných proměnných a poskytovat přesnější informace o ukazatelových proměnných zpět do nástroje EVA. Kombinace obou nástrojů by pak byla schopná analyzovat programy, které ani jeden z nich aktuálně analyzovat nedokáže.

Static Analysis of Heap-Manipulating Programs using Separation Logic

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Dacík. Supplementary information was provided by prof. Ing. Tomáš Vojnar, Ph.D. I have listed all literary sources, publications, and other resources used during the preparation of this thesis.

.....

Tomáš Brablec
May 12, 2025

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Dacík, and my advisor, prof. Ing. Tomáš Vojnar, Ph.D., for introducing me to the topic of software verification and for their guidance during the preparation of this thesis.

Contents

1 Introduction 2
2 Preliminaries 3
2.1 Static Analysis of Programs 3
2.2 Separation Logic 3
2.3 Frama-C 6
3 Existing Approaches 8
4 Analysis 9
4.1 Preprocessing 9
4.2 Dataflow Analysis 9
4.3 Integration with Ivette 14
5 Implementation Details 15
5.1 Preprocessing 15
5.2 Representation of SL Formulae 19
5.3 Interprocedural Analysis 22
5.4 Simplifications 24
5.5 Abstraction 26
5.6 Deduplication Sorting 28
5.7 Configuration 29
6 Results 30
6.1 Crafted Benchmarks 31
6.2 SV-COMP Benchmarks 32
6.3 Evaluation of SL Queries 36
7 Conclusion 38
7.1 Future Work 38

1 Introduction

There are two basic approaches to memory management in programming languages. One approach relies on a garbage collector to reclaim unreachable allocated memory automatically. The other, manual memory management, relies on the programmer to allocate and free memory explicitly. Some languages that rely on manual memory management allow for a group of bugs, called memory safety bugs, to occur. These include *use-after-free* – access to already freed memory, null-pointer dereferences, *double-free* – deallocating already freed memory, or *memory leaks* – a loss of reference to an allocation. These bugs are often a source of security vulnerabilities in affected programs, since invalid memory accesses are undefined behavior in C and C++ [1, 2].

There are a number of techniques to find these bugs and ensure the memory safety of programs. One approach is testing and dynamic analysis, often by code instrumentation done by the compiler. Another approach is static analysis and formal verification. Formal verification is a collection of methods to prove the correctness of a program or its specific property. In this case, the property is memory safety, i.e., that every memory access in the program is valid and memory (de)allocation is done correctly. Methods of static analysis are described in Section 2.1.

One of the approaches to static analysis is so-called *dataflow analysis*, which relies on traversing the control flow graph (CFG) of a program and tracking certain information at each node of the CFG. The *Frama-C framework* described in Section 2.3 provides a generic framework for implementing a dataflow analysis of C programs.

Formulae of *separation logic* are often used to represent the shapes of data structures for the purpose of static analysis. Separation logic allows us to describe structures such as linked lists of unbounded size in a single formula. Models of these formulae then represent concrete configurations of the heap. Separation logic is explained in detail in Section 2.2.

Section 3 describes existing approaches to static analysis of programs with a focus on proving the memory safety of programs. Analyzers that use separation logic are also mentioned.

Section 4 introduces our new static analyzer, KTSN, aimed at verifying the memory safety of C programs. The tool is focused on programs operating on linked lists, as this is a common data structure in low-level software. The analyzer is based on dataflow analysis and uses formulae of separation logic to represent abstract memory states. To determine which states generated during the analysis are fully covered by other states, a solver for separation logic *Astral* is used. This, along with the abstraction of pointer chains into list predicates, is needed to find invariants for loops in the program, which is required for the analysis to terminate.

Section 5 describes the implementation details of the analyzer. KTSN is implemented using the Frama-C framework, which also provides a simplified representation of the input program's AST with an API for its transformation. This is used to preprocess the original AST before the analysis.

In Section 6, the analyzer is tested on crafted programs that work with all supported types of linked lists, trying common list operations such as construction, traversal, insertion into the list, reversal, deallocation, and others. The tool is also tested on a subset of the SV-COMP benchmarks dedicated to linked lists and compared to state-of-the-art tools in this category. The time needed to evaluate queries to the solver during the analysis is also benchmarked.

Section 7 summarizes this work and discusses possible future improvements of the analysis, including the integration of our method into EVA, a value analyzer of the Frama-C framework.

2 Preliminaries

This section describes concepts and technologies needed to understand the design and implementation of our analysis.

2.1 Static Analysis of Programs

Static analysis of programs is a collection of methods to determine some properties of programs without executing them. Static analysis aims to detect bugs in software, ensure code quality, enable optimizations during compilation, or prove the correctness of programs. The main advantage of static analysis compared to dynamic analysis, which involves running the program in different configurations, is that it can reason about all possible program executions at once and can therefore prove properties of programs. Some of the methods used for static analysis fall into the category of formal verification – methods designed to prove the correctness of programs.

There are numerous approaches to formal verification. One of these is *abstract interpretation*, a method that simulates program execution while over-approximating program values on some abstract domain (for example, using intervals for integer values). Formulae of separation logic are one of these abstract domains, especially useful for verifying the memory safety of programs.

Another approach is *model checking*, which verifies that a finite-state model of a system satisfies some specification. A typical use case might be the verification of concurrent systems. Temporal logics are often used for this purpose.

Symbolic execution is another common approach that simulates the execution of a program, but uses symbolic values instead of concrete values, generating constraints for different paths through the program. Solving these constraints can then be used to, for example, discover inputs that trigger bugs in programs.

2.2 Separation Logic

Separation logic (SL) [3, 4] is one of the most popular formalisms used for analysis of programs that manipulate dynamically allocated memory. A formula of separation logic allows us to describe the program's heap so that models of such a formula, called *stack-heap models*, represent concrete structures of allocations and pointers connecting them. The main advantage of separation logic over other formalisms used for describing dynamic memory is the ability to describe disjoint parts of the heap using the *separating conjunction* ($*$) operator. Another key component is a set of *inductive predicates*, which describe data structures of an unbounded size in a closed form.

Due to the high expressivity of general separation logic, existing solvers usually focus on a certain subset of all SL formulae, called *fragments*. One of these fragments commonly used for static analysis is called the *symbolic heap fragment*, which has a limitation in the use of boolean connectives.

The rest of this section describes a fragment of separation logic called *boolean separation logic* as defined in [5], which, unlike other SL fragments, allows for a limited use of boolean operators – conjunction, disjunction, and guarded negation ($\varphi \wedge_{\neg} \psi$). Boolean connectives are needed for the representation of lists with a non-zero minimum length and for representing a state using a disjunction of multiple formulae.

2.2.1 Syntax

Variables in SL have a *sort* representing the type of list the variable can be a part of. There are three predefined sorts, $\mathbb{S}$ for singly-linked lists (SLS), $\mathbb{D}$ for doubly-linked lists (DLS), and $\mathbb{N}$ for nested singly-linked lists (NLS). We denote $x^{\mathbb{S}}$ as a variable of sort $\mathbb{S}$. Additionally, generic sorts can be defined along with custom fields $\mathrm{f}_1, ..., \mathrm{f}_\mathrm{n}$ for its points-to predicate. In the grammar, these sorts are denoted as $\mathbb{G}$. There is also a special variable `nil` without a sort.

The syntax of SL formulae φ is described by this grammar:

$$
\begin{aligned}
\pi_p &:= x^{\mathbb{S}} \mapsto n \\
&\mid x^{\mathbb{D}} \mapsto \langle \mathrm{n} : n,\ \mathrm{p} : p \rangle \\
&\mid x^{\mathbb{N}} \mapsto \langle \mathrm{t} : t,\ \mathrm{n} : n \rangle \\
&\mid x^{\mathbb{G}} \mapsto \langle \mathrm{f}_1 : f_1,\ ...,\ \mathrm{f}_\mathrm{n} : f_n \rangle && \text{(points-to predicates)} \\
\pi_l &:= \mathtt{ls}\big(x^{\mathbb{S}}, y^{\mathbb{S}}\big) \mid \mathtt{dls}\big(x^{\mathbb{D}}, y^{\mathbb{D}}, p^{\mathbb{D}}, n^{\mathbb{D}}\big) \mid \mathtt{nls}\big(x^{\mathbb{N}}, y^{\mathbb{N}}, z^{\mathbb{S}}\big) && \text{(list predicates)} \\
\varphi_a &:= x = y \mid x \neq y \mid \mathtt{freed}(x) \mid \mathtt{emp} \mid \pi_p \mid \pi_l && \text{(atomic formulae)} \\
\varphi &:= \varphi_a \mid \exists x.\ \varphi \mid \varphi * \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge_{\neg} \varphi && \text{(formulae)}
\end{aligned}
$$

We write $\varphi[b/a]$ for the formula φ in which all occurrences of a are replaced by b.

2.2.2 Semantics

Let `Vars` be a set of variables, `Locs` a set of locations, and `Fields` a set of fields. The model of an SL formula, called a stack-heap model, is denoted as a pair (s, h). The *stack* is a partial function $\mathtt{Vars} \to \mathtt{Locs}$, and the *heap* is a partial function $\mathtt{Locs} \to (\mathtt{Fields} \to \mathtt{Locs})$. The partial function $(\mathtt{Fields} \to \mathtt{Locs})$ can be written as an enumeration of all fields and their corresponding locations, for example $h(\ell) = \langle \mathrm{f}_1 : \ell_1,\ \mathrm{f}_2 : \ell_2 \rangle$. Let $\mathtt{dom}(h)$ be the domain of the partial function h.

The semantics of the special variable `nil` is that $s(\mathtt{nil}) \notin \mathtt{dom}(h)$. There is a special location $\boxtimes \in \mathtt{Locs}$, used in the definition of the `freed` predicate. Its semantics are the same as for `nil`, $\boxtimes \notin \mathtt{dom}(h)$.

The semantics of boolean separation logic is defined in Listing 1.

Equality and inequality of variables are defined as expected, but note that they require the heap to be empty. Therefore, separating conjunction is used to join all atomic formulae instead of regular conjunction. For example, a heap with one allocation will be described as $x \mapsto y * y = \mathtt{nil}$. The formula $x \mapsto y \wedge y = \mathtt{nil}$ would be unsatisfiable, since the atom $y = \mathtt{nil}$ describes an empty heap.

The $\mathtt{freed}(x)$ atom says that the variable x is not allocated. It has been added to Astral for the purpose of our analysis to explicitly mark freed variables. The key property is that adding $\mathtt{freed}(x)$ to a formula containing x discards its models where x aliases with another allocated variable. For example, the formula $x \mapsto y$ has two models, one where x and y are distinct locations and another where they alias:

$$s_1 = \{x \mapsto \ell_0, y \mapsto \ell_1\}, h_1 = \{\ell_0 \mapsto \langle \mathrm{n} : \ell_1 \rangle\}$$

$$s_2 = \{x \mapsto \ell_0, y \mapsto \ell_0\}, h_2 = \{\ell_0 \mapsto \langle \mathrm{n} : \ell_0 \rangle\}$$

Changing the formula to $x \mapsto y * \mathtt{freed}(y)$ removes the second model because it adds the implied inequality $x \neq y$. The `freed` atom works the same for a variable of any sort; the specific fields in the semantics of the `freed` atom are irrelevant.

$(s,h) \vDash x = y$ iff $s(x) = s(y)$ and $\texttt{dom}(h) = \emptyset$

$(s,h) \vDash x \neq y$ iff $s(x) \neq s(y)$ and $\texttt{dom}(h) = \emptyset$

$(s,h) \vDash \texttt{freed}(x)$ iff $h = \{s(x) \mapsto \langle \mathrm{n} : \boxtimes \rangle\}$

$(s,h) \vDash \texttt{emp}$ iff $\texttt{dom}(h) = \emptyset$

$(s,h) \vDash x \mapsto \langle \mathrm{f}_1 : f_1, \ldots, \mathrm{f}_n : f_n \rangle$ iff $h = \{s(x) \mapsto \langle \mathrm{f}_1 : f_1, \ldots, \mathrm{f}_n : f_n \rangle\}$
and $s(f_i) \neq \boxtimes$ for each i

$(s,h) \vDash \varphi_1 \wedge \varphi_2$ iff $(s,h) \vDash \varphi_1$ and $(s,h) \vDash \varphi_2$

$(s,h) \vDash \varphi_1 \vee \varphi_2$ iff $(s,h) \vDash \varphi_1$ or $(s,h) \vDash \varphi_2$

$(s,h) \vDash \varphi_1 \wedge_\neg \varphi_2$ iff $(s,h) \vDash \varphi_1$ and $(s,h) \nvDash \varphi_2$

$(s,h) \vDash \exists x.\ \varphi$ iff there is a location ℓ such that $(s[x \mapsto \ell], h) \vDash \varphi$

$(s,h) \vDash \texttt{ls}(x,y)$ iff $(s,h) \vDash x = y$, or
$s(x) \neq s(y)$ and $(s,h) \vDash \exists x'.\ x \mapsto x' * \texttt{ls}(x', y)$

$(s,h) \vDash \texttt{dls}(x,y,p,n)$ iff $(s,h) \vDash x = n * y = p$, or
$s(x) \neq s(n), s(y) \neq s(p), s(p) \notin \texttt{dom}(h)$, and
$(s,h) \vDash \exists x'.\ x \mapsto \langle \mathrm{n} : x', \mathrm{p} : p \rangle * \texttt{dls}(x', y, x, n)$

$(s,h) \vDash \texttt{nls}(x,y,z)$ iff $(s,h) \vDash x = y$, or
$s(x) \neq s(y)$ and
$(s,h) \vDash \exists t', n'.\ x \mapsto \langle \mathrm{t} : t', \mathrm{n} : n' \rangle * \texttt{ls}(n', z) * \texttt{nls}(t', y, z)$

Listing 1: The semantics of separation logic

The semantics of the points-to predicate and boolean atoms is as expected. Note that the definition of the points-to predicate guarantees that no atom can point to the special location $\boxtimes$. Otherwise, $x \mapsto y \wedge \texttt{freed}(x)$ would have a model where $s(y) = \boxtimes$.

The inductive predicates describe chains of pointers of an unbounded length:

- $\texttt{ls}(x,y)$ describes a singly-linked acyclic list of length zero or more. In the first case, the list is equivalent to $x = y$. Length one is equivalent to a simple pointer $x \mapsto y * x \neq y$. All other models contain unnamed allocated memory locations in the chain between $s(x)$ and $s(y)$. Note that y itself is not allocated in any model.
- $\texttt{dls}(x,y,p,n)$ represents a doubly-linked acyclic list of allocations. Unlike in the previous case, both x and y are allocated. The variables p and n represent the `p` and `n` fields of the first and last allocation, respectively. Similar to the singly-linked list, the zero-length case is equivalent to $x = n * y = p$, and the length one is equivalent to a single points-to atom.
- $\texttt{nls}(x,y,z)$ describes a nested acyclic list. The top-level list leads through the `t` field from x to y, and there is a sublist from each node's `n` field leading to z. The sublist itself is not a nested list, but a singly-linked list. Like the other lists, NLS can be empty (equivalent to $x = y$), but note that even the case of length one contains a sublist of an unbounded length.

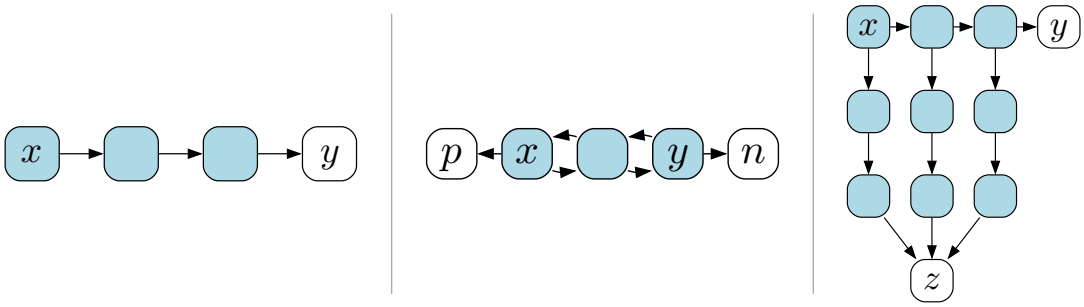


Figure 1: Supported types of lists with the allocated nodes colored blue: singly-linked list, doubly-linked list, nested list

2.3 Frama-C

Frama-C [6, 7] is a framework for building analysis tools aimed at C programs. Frama-C itself is written mainly in the OCaml programming language. Unlike other tools, which focus purely on finding bugs using heuristics, Frama-C specializes in verification. The framework itself is composed of a kernel, multiple plugins, and a GUI to present the results of analyses. The kernel provides common functionality for multiple plugins. The main component is an adapted form of the *C Intermediate Language* (CIL) [8], constructed by the Frama-C kernel for use within plugins, as well as an API for its manipulation. CIL has the form of an abstract syntax tree (AST) of the input source code, with extra semantic information added. This includes types of variables, whether a variable is initialized at a certain node, and other information. Frama-C also transforms the input code, making operations like type casts explicit and otherwise making the code more suitable for static analysis. For example, all `return` statements in a function are replaced with `goto` statements that lead to a single `return` at the end of the function. `for` and `while` loops are replaced with a simple infinite loop with an exit condition inside.

One of the goals of Frama-C is to help with the development of custom analyses in the form of plugins. Frama-C handles command-line argument parsing, reports analysis results, and exposes APIs for access to the input CIL.

2.3.1 Dataflow Analysis

Besides a general framework for implementing analyses, Frama-C provides generic implementations of common algorithms used for static analysis. One of these is dataflow analysis, implemented in module `Dataflow2`. The purpose of dataflow analysis is to compute certain information for each node of the control flow graph (CFG) in the analyzed program. Nodes in a control flow graph correspond to statements in the original program, and directed edges represent all possible jumps between these statements.

Dataflow analysis starts by assigning an initial state to each node of the CFG, and then progressively updates the values stored in nodes using an implementer-provided *transfer function*, following the edges between them. The transfer function takes the statement of the node and the newly calculated state of the previous node, and produces a new state for the current node. In Frama-C, this function is called `doInstr`.

When a node is reached more than once, the data for this node is computed again based on the data from the previous node, and then joined with the previous data stored for the node. The function that joins the old state with the new state is also provided by the implementer. The name of this function in Frama-C is `combinePredecessors`.

The implementer must also provide the following functions:

- `doStmt` – This function is called before the transfer function. The plugin has the option to stop the analysis of this statement or continue normally.
- `doGuard` – This function is called when a conditional statement is reached. It receives the state from the previous node and the condition expression, and generates two states, each to be used in one of the branches.
- `doEdge` – This function is called between the analysis of two statements. The function receives both statements and the current state of the first statement. The function can modify this state before continuing with the transfer function of the second statement.

This list is not exhaustive, but the implementation of other functions needed by the `Dataflow2` module is not relevant to the analysis itself.

2.3.2 Visitor Mechanism

Frama-C provides a convenient way to modify the AST of the analyzed program using the visitor pattern. The plugin constructs an object and defines methods corresponding to the AST node it wants to visit. The method decides whether to leave the node as is, change it, or continue with the visits of its children. This visitor object is then applied to the AST. It is possible to visit AST nodes for expressions, statements, variables, types, and other constructs.

2.3.3 Ivette

Ivette [9] is a desktop application for displaying the results of analyses, for which it uses a client-server architecture. Ivette, the client, asynchronously polls the server (Frama-C plugin) for data and displays it. The server first has to register the data that has to be shown. To achieve real-time display of information when running an analysis, the plugin must occasionally call `Async.yield` to let Ivette synchronize the data in the GUI application with the data generated by the running analysis. The API for registering and using Ivette can be found in the module `Server`, inside the library `frama-c-server.core`.

3 Existing Approaches

This section lists some static analysis tools that focus on the memory safety of programs with dynamically allocated data structures, and on tools that use separation logic in their analysis. Both research prototypes and tools used in production are mentioned.

Predator [10] is a verification tool aimed at programs with dynamic data structures. It supports many variants of lists: singly and doubly-linked, nested, cyclic, and others. It uses abstract interpretation over *Symbolic Memory Graphs* as its domain. The analyzer is able to prove the memory safety of programs – it proves the absence of invalid pointer dereferences, double-free bugs, and memory leaks. PredatorHP, a parallelized version of the tool running in different configurations, consistently dominates the `LinkedLists` subcategory of the SV-COMP competition (see Section 6). Recently, the SMG abstraction used by Predator was reimplemented [11] in the CPAchecker analyzer [12].

EVA [13] is a verifier based on the Frama-C framework, which uses abstract interpretation to detect many kinds of undefined behavior in C programs. It is a general-purpose verifier that tracks the values of both numeric and pointer variables. It can detect invalid memory accesses such as null-pointer dereferences or buffer overruns, reads of uninitialized memory, integer overflows, divisions by zero, and other errors. Its abstract domain uses enumerations and intervals for numeric values and sets of addresses for pointers. However, it does not have the capability to represent the shape of dynamic data structures.

Infer [14] is a production-grade verifier focusing on the memory safety of programs. It uses bi-abductive analysis to infer pre-conditions and post-conditions for functions without the full context of the program, and utilizes separation logic to represent abstract memory states. It uses interprocedural analysis to analyze large codebases and can analyze dynamic data structures such as many variants of lists and trees.

Broom [15] is a verification tool based on bi-abduction and separation logic aimed at verifying the memory safety of low-level programs. It is focused on programs working with dynamic data structures that use advanced pointer-manipulating operations, such as pointer arithmetic, address alignment, and other pointer operations that often appear in the Linux kernel. The tool is able to analyze freestanding code fragments without knowing their full context, which improves the scalability of the analysis.

4 Analysis

The analysis method is an extension of the approach described in [16] and is inspired by the approach in [17]. The analysis itself consists two main steps. First, the analyzed program is preprocessed into a form more suitable for analysis. Then, the dataflow analysis over the modified AST is executed.

4.1 Preprocessing

The preprocessing starts with an external semantic constant propagation pass with loop unrolling. The purpose of this is to avoid a problem with too loose bounds on the lengths of lists, described in Section 5.1.3.

The next few preprocessing passes convert the program's instructions into a small set of *basic instructions*. *Instruction* is any statement that does not affect the control flow of the current function. The specific preprocessing passes are described in Section 5.1.2. This is the set of basic instructions, for which the rest of our analysis is implemented:

- `_const = var;` – assertion that `var` is an allocated variable (`_const` is only a placeholder),
- `var = var2;` – assignment of a variable into a variable,
- `var = var2->field;` – assignment of a variable's field dereference into a variable,
- `var->field = var2;` – assignment of a variable into a dereferenced field of another variable,
- `*var = var2;` – assignment of a variable into a pointer dereference,
- `var = *var2;` – assignment of a pointer dereference into a variable,
- `var = &var2;` – assignment of a reference to a variable into a variable,
- `fun(var1, ..., varN);` or `var = fun(var1, ..., varN);` – function call with an optional assignment into a variable, with all arguments being variables.

This is followed by an analysis of C types. As described in Section 2.2, when introducing a variable into a formula, we need to provide its sort. In our analysis, this sort is derived from the type of the corresponding C variable. For types that form one of the supported kinds of lists, a special sort predefined by Astral must be used, so that points-to atoms from these variables can later be abstracted to list atoms. For all other structure types, a new sort must be declared and registered in the solver instance. See Section 5.1.6 for a detailed description.

4.2 Dataflow Analysis

The analysis starts at the first statement of the `main` function with `emp` as the initial state, and runs the dataflow analysis from this statement. The domain of the dataflow analysis is the disjunction of symbolic heaps. Symbolic heap is a formula in the form of $\varphi_1 * ... * \varphi_n$, where each φ_i is an atomic formula. The formulae used by the analysis are not actually represented by the types provided by Astral, but instead use a custom, flattened formula type that is simpler to work with. The details are described in Section 5.2.

Variables in formulae have two variants, *program variables* and *logic variables*. Program variables directly correspond to C variables currently in scope with the same name. Logic variables correspond to memory locations that do not currently have a name in the analyzed program. In SL formulae, they are implicitly existentially quantified. More on this in Section 4.2.3. In this text, we refer to logic variables using "primed notation", i.e., as f'_n.

The analysis itself is implemented using the `Dataflow2` module provided by Frama-C (see Section 2.3.1), and it is composed of four main components. First, there is the transfer function implemented for the basic instructions described above. Then, there are the simplifications applied to the formulae between instructions. The abstraction (over-approximation of pointer

sequences by list predicates) is included in the simplifications. There is also the logic for the join operation. Finally, there is the specialization of formulae for each branch of a condition.

4.2.1 Transfer Function

The transfer function takes formulae describing the state before executing an instruction and changes them to reflect the state after the execution. The transfer function always processes the input formulae one by one, applying the operation described below to each input formula (here called φ) separately. When the materialization of a variable x is done, the list predicate starting at x is split into a points-to atom from x and the rest of the list. If x is not allocated in a formula during materialization, an invalid dereference is reported. The transfer function is implemented for each of the basic instructions:

- Assignment into the special `_const` variable `_const = a;` means that a is checked to be allocated. If not, an invalid dereference is reported. The formula is not changed in any way. This corresponds to accesses to non-pointer fields in the original source code, see Section 5.1.1 for more details.
- For simple assignment `a = b;`, the resulting formula is $\varphi[f'_0/a] * a = b$ where f'_0 is a new logic variable. The renaming is done because the memory location originally referred to by a still has to be represented in the formula.
- For an assignment with a field access on the right-hand side `a = b->f;`, the variable b is first materialized in the formula, and then the target of b at field `f` is found (here called t_b). Then, a regular assignment is done with t_b as the assigned value. The resulting formula is $\varphi[f'_0/a] * a = t_b$.
- When processing an assignment to a field `a->f = b;`, a is first materialized, and then the field `f` of the points-to atom from a is changed to b. No substitution is done in this case because we are not overwriting the value of any variable.
- `a = *b;` is processed the same way as `a = b->f;`, except that the "field" being dereferenced has a special name `_target`. Pointers to pointers are represented as pointers to structs with a single field `_target`.
- Since our analysis assumes that pointers to pointers are allocated on the stack, `*a = b;` is interpreted as an assignment of b into the variable pointed to by a. The target of a, called t_a is found, and the formula is transformed as a regular assignment of b into it. After that, the points-to atom from a is changed back to $a \mapsto \langle \_\text{target} : t_a \rangle$ (it was affected by the renaming done during the assignment).
- `a = &b;` is processed differently based on whether a is already allocated (there is a points-to atom from a already in the formula). If so, its `_target` field is simply set to b. If not, this atom is added. This is done to prevent adding a second points-to atom from a, which would make the formula unsatisfiable.

The last instruction is a function call. If the function is from the allocation API (`malloc`, `calloc`, and `free`), it is handled differently from a user-defined function.

- Processing `a = malloc(size);` yields two formulae, one representing the successful allocation with its fields set to new logic variables: $\varphi * a \mapsto \langle \text{f}_0 : f'_0, \text{f}_1 : f'_1, ... \rangle$, the other one representing an allocation failure: $\varphi * a = \texttt{nil}$.
- `a = calloc(size);` is handled the same as `malloc`, except that all fields of the allocated points-to are set to `nil` instead of new logic variables.

- `free(a);` is handled by materializing a and then removing the points-to atom starting at a. If the materialization fails, an invalid free is reported instead of an invalid dereference.

All other function calls are handled explicitly by splitting the formula into two parts by reachability from the function arguments, and then running the dataflow analysis recursively on the called function, using the reachable subformula as the initial state. The resulting state of analyzing the function is taken from its return statement. To improve the performance, a cache of *function summaries* is used. The input to the cache is a modified initial state, the output is a modified output state. The implementation of the cache is described in Section 5.3.

4.2.2 Simplifications

Between analyzing instructions, the state formulae are simplified and abstracted to ensure the termination of the analysis. The reasoning behind some of these simplifications is provided in Section 5.4. The following simplifications are applied in this order:

- Program variables going out of scope by the transition between the two instructions are substituted with logic variables.
- Points-to atoms representing pointers to variables going out of scope are removed.
- Spatial atoms unreachable from program variables are removed and reported as memory leaks.
- Logic variables equivalent to program variables are removed from the formula by substituting them with these program variables.
- Chains of spatial predicates are abstracted to list predicates. This step ensures the analysis of loops will terminate. More on this in Section 5.5.
- Inequalities and `freed` atoms containing logic variables only present in these atoms are removed. Such atoms do not hold any information useful for the analysis.
- List atoms known to be empty are removed or replaced with equivalences. These are the products of previous simplifications and must be removed to avoid accumulating meaningless atoms in formulae. List atoms are assumed to be empty when the start and the end of the list are syntactically equivalent in the formula.
- The formulae are canonicalized, which involves sorting variables in equivalence classes and atoms in formulae. This is done to make the next steps more likely to succeed.
- Formulae differing only in the length bound of a single spatial atom are merged, taking the lower bound of the two as the new bound.
- Formulae are deduplicated by syntactical comparison, and then semantically using the procedure described in Section 4.2.3.

4.2.3 Join Operation

When the analysis reaches a statement for the second or subsequent times, a state φ_{old} is already associated with the statement. A new state φ_{new} is generated using the transfer function as usual, and then these states are joined to produce either a new, combined state to be stored for the statement, or nothing, if the new state is covered by the original state. By covering, we mean that all models of the new state are already included in the models of the original state. In other words, it must hold that $\varphi_{\text{new}} \vDash \varphi_{\text{old}}$, where φ_{old} and φ_{new} are disjunctions of the individual SL formulae comprising the two states.

The evaluation of satisfiability and entailment checks is cached. As an optimization, an entailment is not evaluated as-is, but is broken down into a series of smaller entailments. This improves the efficiency of the cache because smaller entailments have a higher chance of appearing multiple times during analysis.

The states φ_{old} and φ_{new} are first split into lists of their formulae, and these lists are concatenated. This list of old and new formulae is first sorted (see Section 5.6) and then deduplicated using the following algorithm: a list of deduplicated formulae φ_{out} is created empty, and then, one by one, the formulae are added to this list. Before adding a formula φ_0 into $\varphi_{\text{out}} = [\varphi_1, \varphi_2, ...]$, the entailments $\varphi_0 \vDash \varphi_1$, $\varphi_0 \vDash \varphi_2$, … are checked and only if none of them succeed, φ_0 is added to φ_{out}.

After the deduplication, we must decide if the new state φ_{out} is covered by φ_{old}. Again, the entailment is split into a series of smaller entailments, where each formula from φ_{out} is tested separately. The test of φ_0 from φ_{out} against $\varphi_{\text{old}} = [\varphi_1, \varphi_2, ...]$ is done formula by formula, if at least one of the entailments $\varphi_0 \vDash \varphi_1$, $\varphi_0 \vDash \varphi_2$, … succeeds, φ_0 is covered by the old state. Each formula from φ_{out} is tested this way, and if all of them are covered, the join operation reports that φ_{old} covers the new state, and the analysis does not continue beyond this statement.

4.2.4 Condition Evaluation

When reaching a conditional statement, the analysis must decide whether to visit the positive and negative branches of the conditional, and how to modify the states used in the analysis of these branches. The only analyzed conditions are the equality and inequality of two pointer variables. All other conditions are treated as nondeterministic – both branches are analyzed with the unchanged input state. When a nondeterministic condition is reached, any subsequent detection of error in the program is not reported as an error but as an unknown result. This must be done to prevent false detections of invalid dereferences in programs that use lists of bounded length, as is explained in Section 5.1.3.

When reaching a condition `a == b`, the equality of a and b is added to the formulae for the positive branch using `add_eq` described in Section 5.2.2. Unsatisfiable formulae are then filtered out. If there are any remaining formulae, the analysis continues into the positive branch with these. If not, the analysis does not continue into the branch at all. For the negative branch, the algorithm is the same, only the inequality of a and b is added to the original formulae instead of equality. The function `add_distinct` is used for adding the inequality, as described in Section 5.2.2.

When reaching a condition `a != b`, the approach is the same, except that the positive and negative branches are swapped.

4.2.5 Example

The example in Listing 2 shows how the analysis makes progress on finding a fixpoint of an allocation loop. After three iterations through the loop, the analysis finds the fixpoint to be the following state:

$$\bigvee \left[\begin{array}{c} x \mapsto f_1' * y = x \\ x \mapsto y * y \mapsto f_2' \\ \mathtt{ls}_{2+}(x, y) * y \mapsto f_3' \end{array} \right]$$

Note that after the analysis finds the fixpoint, the state is further simplified to this form through formula generalization:

$$\bigvee \left[\begin{array}{c} x \mapsto f_1' * y = x \\ \mathtt{ls}_{1+}(x, y) * y \mapsto f_2' \end{array} \right]$$

Code	State formulae
`1 Node *x = malloc(size);`	$x \mapsto f_1'$
`2 Node *y = x;`	$x \mapsto f_1' * y = x$
`3 while (rand()) {`	$x \mapsto f_1' * y = x$ $x \mapsto y * y \mapsto f_2'$ $\mathtt{ls}_{2+}(x, y) * y \mapsto f_3'$
`4 y->next = malloc(size);`	$x \mapsto f_1' * f_1' \mapsto f_2' * y = x$ $x \mapsto y * y \mapsto f_2' * f_2' \mapsto f_3'$ $\mathtt{ls}_{2+}(x, y) * y \mapsto f_3' * f_3' \mapsto f_4'$
`5 y = y->next;`	$x \mapsto y * y \mapsto f_2'$ $x \mapsto f_4' * f_4' \mapsto y * y \mapsto f_3'$ $\mathtt{ls}_{2+}(x, f_5') * f_5' \mapsto y * y \mapsto f_4'$
`6 }`	$x \mapsto f_1' * y = x$ $\mathtt{ls}_{1+}(x, y) * y \mapsto f_2'$
`7 y->next = NULL;`	$x \mapsto \mathtt{nil} * y = x$ $\mathtt{ls}_{1+}(x, y) * y \mapsto \mathtt{nil}$

Listing 2: Simplified progress of an analysis run, the color of a formula indicates when it was generated:

- before loop + first loop iteration
- second loop iteration
- third loop iteration + after loop

4.3 Integration with Ivette

The analyzer supports two modes. It can run both in the terminal and in the graphical application *Ivette* provided by Frama-C. Ivette starts before the analysis begins and shows the intermediate results of the analysis as they are generated. The user can see the original program (top left window in the screenshot below) and its preprocessed version (top right window). This was especially useful during development, because by hovering the cursor above a part of the original program, the equivalent part of the preprocessed code is highlighted, and vice versa. The GUI also shows a list of messages emitted by the analyzer (bottom left window). These messages are divided into categories based on their origin and can be filtered. Debug messages are emitted from the transfer function, simplifications, function call handling, Astral queries, and other sources. These messages show the state before and after the given step. When highlighting a statement in the preprocessed AST, the GUI shows the current state associated with the statement (bottom right window).

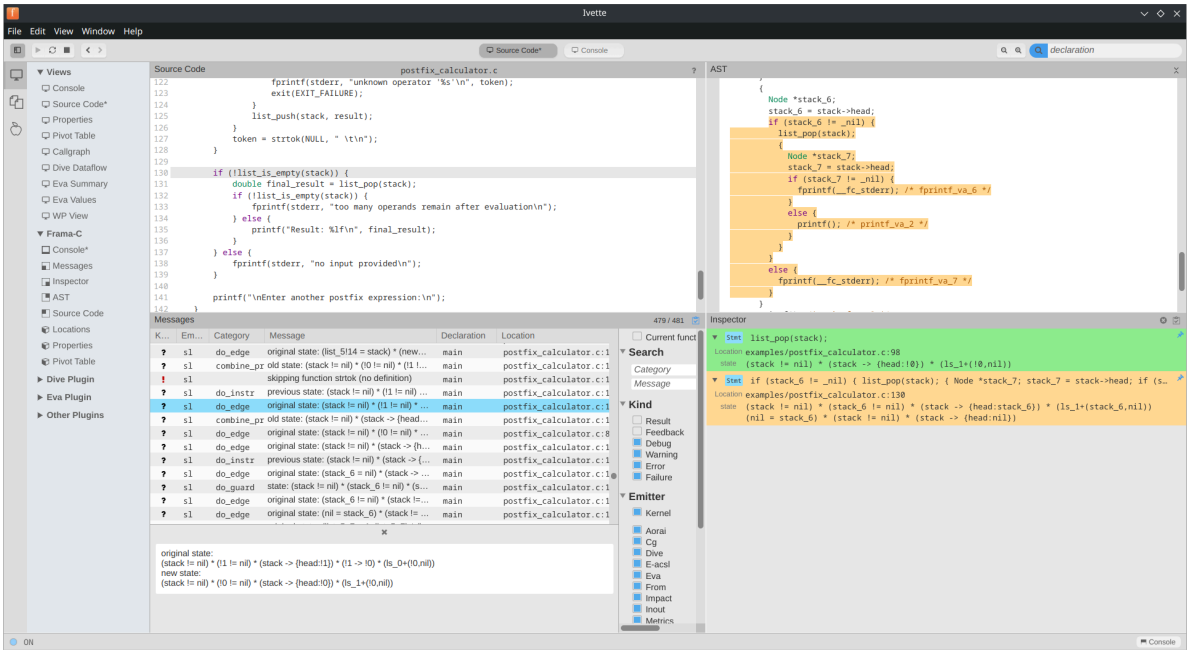


Figure 2: Ivette GUI showing the intermediate results of the analysis

5 Implementation Details

KTSN is implemented in the OCaml programming language. This section describes various aspects of the analysis in more detail, with a focus on how certain parts of our method are implemented in the analyzer.

5.1 Preprocessing

This section describes the details of AST preprocessing done externally by Frama-C (invoked using Frama-C command line parameters) and internally using the CIL Visitor mechanism (see Section 2.3.2). Many of the simple preprocessing passes are implemented directly in `src/preprocessing/preprocessing.ml`. This module also contains the function `preprocess`, which runs all of the preprocessing passes in order. This is called directly from `main` before running the analysis.

5.1.1 Preprocessing of Variables

Some of the preprocessing passes described below replace some expressions with special variables, namely `_nil` and `_const`. These are used to simplify the AST: instead of containing multiple AST nodes such as a constant (zero) or a cast of a constant to a pointer type (`NULL`), all expressions are simply variables. `_nil` is used to represent a `NULL` pointer, whereas `_const` replaces any expression that cannot be analyzed, such as arithmetic expressions or nondeterministic conditions. One other use of `_const` is that assignments into it are seen as an "allocation check" of the variable being assigned. For example, `_const = var;` is not interpreted as a literal assignment into the variable `_const`, but as a check that `var` is allocated. These assignments are created during preprocessing, when a field access to a non-pointer variable is found. For example, the following statement:

```
int data = list->data;
```

will get converted to

```
_const = list;
```

This will not cause any change to the state formulae during the dataflow analysis, but each formula will be checked to ensure that the variable `list` is allocated.

5.1.2 Conversion into Basic Instructions

The conversion of all instructions in the program to basic instructions is done over multiple passes. First, all AST nodes irrelevant for the analysis replaced with `_const`. This includes arithmetic and binary operations, type casts, integer comparison operations, and others. Then, variable initialization is replaced by simple assignments. Next, function call arguments with types irrelevant to the analysis are removed. What constitutes a *relevant type* is defined in Section 5.1.6.

At this point, the AST should ideally only contain analyzable AST nodes. However, it is also necessary to split expressions in assignments, function call arguments, and conditions into a series of elementary assignments. Each complex statement is replaced by a block containing temporary variables, to which parts of the original expression are assigned. This is described in detail in Section 5.1.5.

After this, all assignments into variables with irrelevant types are removed or replaced by a check that the dereferenced variable is allocated. At this point, every instruction in the program should be one of the basic instructions.

5.1.3 Semantic Constant Propagation

The only conditions the analysis is able to process are equalities and inequalities of variables. This turned out to be a problem in many of the SV-COMP benchmarks (for example, in `list-simple/sll2c_update_all.c`), because they often contain manipulation of lists of bounded size. Take the following code as an example:

```
int len = 5;

while (len) {
  append_node(&list, data);
  len--;
}

list->next->next->data = 42;
```

Because our analysis considers all other conditions nondeterministic (see Section 5.1.4), it will over-approximate and include the possibility of zero iterations over this loop. This means that the state after the loop will contain a formula describing a list of zero length. The problem is that the code below the loop will rightfully assume the list has non-zero length, accessing the n-th node in the list without checking that the access is valid. The analysis will then detect a potential invalid dereference, and because a nondeterministic condition had been reached, it will return an unknown result.

The solution to this is to run an analysis of integer values before our analysis and statically determine the number of iterations over the loop. Analysis of numeric values is already implemented in Frama-C in the *Semantic constant folding* (SCF) plugin [18], which in turn uses the EVA [19] plugin for value analysis. This is used in combination with Frama-C's loop unrolling setting `ulevel` to unroll a few iterations of every loop. This preprocessing pass will produce roughly the following code from the example above, when used with the setting `-ulevel=2`:

```
int len = 5;

if (!5)
    goto unfolding_2_loop;
append_node(&list, data);
len = 4;

if (!4)
    goto unfolding_2_loop;
append_node(&list, data);
len = 3;

while (len) {
    append_node(&list, data);
    len--;
}

unfolding_2_loop:;

list->next->next->data = 42;
```

As you can see, the first two iterations of the loop have been unrolled above the loop body, and the variable `len` inside the unrolled conditional jumps was replaced by its value at that moment by the SCF plugin. These constant conditions are then removed altogether, see Section 5.1.4.

5.1.4 Preprocessing of Conditions

Conditions are preprocessed in the following passes:

- Removing constant conditions – this is done before the removal of all constants from the AST, and therefore the conditions left by the SCF pass (see Section 5.1.3) are still intact. Using the `remove_const_conditions` visitor in `src/preprocessing/preprocessing.ml`, all condition expressions that can be folded to a constant using `Cil.constFoldToInt` are evaluated and removed from the code, replacing the original `If` AST node with the `Block` node of the branch that would be executed. This is especially useful for unfolded loop iterations.
- Removing the `!` operator – using the visitor `remove_not_operator` in `src/preprocessing/preprocessing.ml`, the following replacements are made:
  - negation of negation is replaced by the inner expression,
  - negation of equality is replaced by inequality,
  - negation of inequality is replaced by equality.

  These replacements are made because the evaluation of many macros in the SV-COMP benchmarks creates exactly these types of conditions. Simply removing them is easier than implementing the analysis to cover them.
- Splitting the condition expressions – the condition expression is split the same way as other expressions described in Section 5.1.5. This pass is implemented in `src/preprocessing/condition_split.ml`. This pass also converts implicit pointer comparisons to explicit ones. Specifically, these replacements are made:
  - `(var)` is converted to `(var != _nil)`,
  - `(!var)` is converted to `(var == _nil)`.

  Conditions that cannot be preprocessed into a pointer comparison are replaced with the special constant `_const` instead. These are then considered nondeterministic during analysis.

After these passes, all analyzable conditions have the form of an equality or an inequality of two pointer variables. All other conditions simply contain the constant `_const`.

5.1.5 Splitting Complex Statements

To split any supported instruction into a series of basic instructions, this pass must introduce new variables to hold temporary values. Because the analysis handles program and logic variables differently (see Section 5.2), it is more convenient to give these variables as short a lifetime as possible. Each split statement is therefore replaced with a block that holds the necessary temporary variables.

The preprocessing pass itself is implemented in `src/preprocessing/stmt_split.ml`, but the logic behind recursively splitting a single expression into a series of simple assignments, yielding a single variable, is implemented in `src/preprocessing/block_builder.ml`. This logic is also used by the preprocessing pass for conditions, see Section 5.1.4.

The splitting itself is done by storing the result of every single dereference into a temporary variable and using the previously created temporary variables in the latter assignments. For example, the following statement:

```
var1->field1->field2 = (*var2)->field3;
```

will be split into this block:

```
{
  tmp1 = *var2;
  tmp2 = tmp1->field3; // tmp2 contains the value of the original expression
  tmp3 = var1->field1;
  tmp3->field2 = tmp2; // this line performs the original assignment
}
```

Note that it is not possible to extract the last l-value `tmp3->field2` into a temporary variable, and to assign `tmp2` into this variable, because it would not change the value in memory pointed to by the `tmp3` pointer.

5.1.6 Analysis of C Types

The purpose of this analysis is to determine which C types might implement one of the supported list variants, and to store this information for later use. Thanks to this, our analysis does not need any kind of struct annotation. This analysis pass is implemented in `src/preprocessing/types.ml`.

The only *relevant types* for the analysis are pointers to structures (`struct X *`). Pointers to these pointers (`struct X **`, `struct X ***`, etc.) are also supported, but only as pointers to variables on the stack. This is a limitation given only by the current implementation, as the support for pointers to variables on the stack was added late during development because many SV-COMP tests relied on this feature. However, SV-COMP does not contain any tests where pointers to pointers are allocated on the heap, so there was little reason to work on this specific feature over others.

The pass first iterates through all structure types in the code and applies the following heuristic to determine if it implements of the supported linked lists. The result of this heuristic is one of:

- singly-linked list,
- doubly-linked list,
- nested list,
- generic struct.

The algorithm of the heuristic is the following: first, all relevant fields of a structure are recursively analyzed to get their heuristic result. Then:

- If a structure contains exactly one pointer to the structure itself and no pointers to structures marked as a singly-linked list, it is itself marked as a singly-linked list.
- If a structure contains two pointers to itself and no pointers to singly-linked lists, it is marked as a doubly-linked list.
- If a structure contains one pointer to itself and one pointer to a singly-linked structure, it is marked as a nested list.
- Otherwise, it is marked as a generic structure. A new sort and Astral struct definition is created.

Then, each type is connected to its corresponding sort and struct definition in the `type_info` table. The struct definition contains the names and sorts of all fields in the structure.

This method of analysis has a drawback: not all implementations of supported linked lists will use one of these structures. For example, in the SV-COMP benchmark `forester-heap/sll-01-1.c`, the nested list structure is defined as follows:

```
typedef struct TSLL
{
  struct TSLL* next;
  struct TSLL* inner;
} SLL;
```

During the program's runtime, the same structure is used to represent both the top-level list and the nested sublists, which always set their `inner` field to NULL. Since this will be wrongly detected as a doubly-linked list, the analysis will not be able to do abstraction on the generated formulae. Fixpoint for the list creation loop will not be found, and the analysis will timeout.

There are a few possible ways to fix this. One option would be to rely on the user to provide the correct list type and field types manually using Frama-C annotations. Another option might be to detect the correct list type dynamically during analysis, based on the shapes of the data structures that are appearing in the formulae.

5.2 Representation of SL Formulae

The types used to represent formulae in Astral have a tree structure that requires pattern matching to access every term. This makes sense for a solver that needs to support formulae of all shapes on its input, but for the purpose of our analysis, we need a flattened representation of the exact shape of the formulae we use. This will avoid having to pattern-match terms for which we already know their type, and it will allow us to define shorthands for predicates that have to be expressed by a more complex structure in Astral's types. The formula type, along with a series of functions for manipulating these formulae, is implemented in `src/formula.ml`.

The type itself is defined as follows:

```
type var = SL.Variable.t
type ls = { first : var; next : var; min_len : int }
type dls = { first : var; last : var; prev : var; next : var; min_len : int }
type nls = { first : var; top : var; next : var; min_len : int }

type pto_target =
  | LS_t of var
  | DLS_t of var * var
  | NLS_t of var * var
  | Generic of (string * var) list

type atom =
  | Eq of var list
  | Distinct of var * var
  | Freed of var
  | PointsTo of var * pto_target
  | LS of ls
  | DLS of dls
  | NLS of nls

type formula = atom list

type state = formula list
```

State (type used by the dataflow analysis to represent the state in each CFG node) is a list of formulae, and each formula is a list of atoms. `atom` is an enumeration of the predicates used by the analysis.

Variables used by the formulae are represented using the original type from Astral. Program variables have the same name as their corresponding C variables, while logic variables are distinguished by names containing the character `!`. When adding a logic variable to a formula, we need to give it a fresh name that does not appear anywhere else in the formula.

5.2.1 Spatial Atoms

`PointsTo` represents all variants of points-to predicates. Because we need to distinguish between pointers of the three basic list types and all other pointers, the targets of the pointer are represented in a separate structure `pto_target` with the three basic list variants and a `Generic` variant that simply associates field names with target variables.

List predicates are represented using three atoms, with a *length bound* `min_len` indicating the minimum length of the list predicate. For singly-linked lists and nested lists, the maximum valid bound is 2+, and for doubly-linked lists, it is 3+. This is given by the limitations of encoding list predicates into Astral formulae, see Section 5.2.3.

The module also provides functions for manipulating spatial atoms, such as finding and changing the target of a points-to atom in a formula, checking that a variable is allocated in a formula, or converting points-to atoms to lists. The type `field_type` is used to represent the type of a field in a points-to atom. It is defined in `src/preprocessing/types.ml` in the following way:

```
type field_type = Next | Prev | Top | Other of string | Data
```

The variant `Other` is used to represent an arbitrary field using its name.

Another operation implemented for formulae is the *materialization* of a variable inside a formula. This operation unrolls a single points-to predicate from a list predicate, so that subsequent operations can work with the points-to atom directly. By definition, it asserts that the materialized variable is allocated. Materialization involves adding a logic variable serving as the new start of the list predicate. When materialization is performed on a variable that is already a source of a points-to atom, the formula is left unchanged.

The function `materialize` is implemented recursively and returns a list of formulae because materialization can produce more than one formula. When materializing a list with the length bound of one or higher, the length bound is simply decremented, and the unfolded points-to atom is added to the formula. When the list atom has the length bound 0+, a case split is performed:

- One option is for the list to have a minimum length of at least one. In this case, the length bound is incremented, and `materialize` is called recursively on this modified formula.
- The other option is for the list to have a length of zero. In this case, it is replaced with an equivalence, and `materialize` is called recursively.

The results of evaluating both cases are then concatenated and returned as a list of materialized formulae.

For example, materializing the variable x in $\texttt{ls}_{2+}(x, y)$ yields a single formula $x \mapsto f'_0 * \texttt{ls}_{1+}(f'_0, y)$. Materializing x in $\texttt{ls}_{0+}(x, y) * \texttt{ls}_{1+}(y, z)$ yields a list of two formulae:

$$x \mapsto f'_0 * \texttt{ls}_{0+}(f'_0, y) * \texttt{ls}_{1+}(y, z)$$
$$x = y * x \mapsto f'_0 * \texttt{ls}_{0+}(f'_0, z)$$

The first formula represents the case where the original list $\texttt{ls}_{0+}(x, y)$ had a length of at least one. The second formula represents the case of the list being empty. Note that the materialization also explicitly makes x the source of the unfolded points-to atom.

Materialization of DLS variables is done similarly, with the difference that DLSs can be materialized from any side. In $\texttt{dls}(x, y, p, n)$, either x or y can be materialized. During the materialization in NLSs, an additional logic variable representing the start of the SLS sublist is introduced. For example, materializing x in $\texttt{nls}_{1+}(x, y, z)$ yields

$$x \mapsto \langle \mathrm{t} : f'_0,\ \mathrm{n} : f'_1 \rangle * \texttt{ls}_{0+}(f'_1, z) * \texttt{nls}_{0+}(f'_0, y, z)$$

where f'_1 is the additional SLS variable.

5.2.2 Equality and Inequality

`Eq` represents a list of variables that form an equivalence class. Equivalences in formulae are not handled arbitrarily as pairs of equivalent variables, but a minimal list of disjoint equivalence classes is maintained throughout the analysis. When adding a new equivalence into a formula, the `add_eq lhs rhs` function is used. This function first checks if `lhs` or `rhs` is present in any of the existing equivalence classes, and then:

- If both variables are already in a single equivalence class, nothing happens.
- If both variables are in different equivalence classes, these classes are merged into one.
- If just one variable is in an existing equivalence class, the other variable is added to it.
- Otherwise, a new equivalence class with the two variables is added to the formula.

`Ne` represents the inequality of two variables. Unlike for equality, there is no need to represent a list of variables all distinct from each other. Nonetheless, there is a function `add_distinct lhs rhs` for adding inequalities with special behavior. If there is a list atom in the formula with its source and destination equivalent to `lhs` and `rhs` of the inequality, the length bound of the list is increased to reflect this, instead of adding an explicit inequality. Specifically,

- If $x \neq y$ is added to $\texttt{ls}_{0+}(x, y)$, it becomes $\texttt{ls}_{1+}(x, y)$.
- If $x \neq n$ or $y \neq p$ is added to $\texttt{dls}_{0+}(x, y, p, n)$, it becomes $\texttt{dls}_{1+}(x, y, p, n)$.
- If $x \neq y$ is added to $\texttt{dls}_{1+}(x, y, p, n)$, it becomes $\texttt{dls}_{2+}(x, y, p, n)$.
- If $x \neq y$ is added to $\texttt{nls}_{0+}(x, y, z)$, it becomes $\texttt{nls}_{1+}(x, y, z)$.

This comes from the fact that all zero-length list predicates require the source and the target to be equal. By adding this inequality, we essentially restrict the possible cases to those where the list has a length of at least one. Similarly, in the third case, if the first and last allocated locations in a DLS differ, the list must have a length of at least two.

The reason for doing this is to prevent false detections of invalid dereferences when analyzing code that asserts a list to be non-empty using a condition, and then accesses this list. For example, consider the following code:

```
a = construct_list();
if (a != NULL) {
  a->data = 42;
}
```

The state after the call to `construct_list` might be $\texttt{ls}_{0+}(a, \texttt{nil})$. When analyzing the condition, the inequality $a \neq \texttt{nil}$ must be added. If we simply added the inequality atom to the formula, we would detect a possible invalid dereference at the assignment. The code checking that a variable is allocated simply materializes the variable and checks that there is a spatial atom with its

source equivalent to that variable. This check would fail for the formula $\texttt{ls}_{0+}(a, \texttt{nil}) * a \neq \texttt{nil}$. Increasing the bound on the list predicate solves this issue.

5.2.3 Translation to Astral Types

The translation of equality, inequality, and `freed` atoms to the types of Astral's formulae is trivial, as is the translation of points-to atoms of the three predefined list types. Translating points-to atoms with sorts defined at runtime involves adding a struct definition for the variable's sort that was created during the analysis of types (see Section 5.1.6).

Translating list predicates involves expressing the length bounds implicitly, because Astral does not have a way to explicitly set the minimum length of a list predicate. Therefore, only $\texttt{ls}_{0+}(x, y)$, $\texttt{dls}_{0+}(x, y, p, n)$ and $\texttt{nls}_{0+}(x, y, z)$ are translated directly to their corresponding list predicates in Astral's types. The rest is translated recursively according to the following rules:

$$
\begin{aligned}
\texttt{ls}_{1+}(x, y) &\rightarrow \texttt{ls}_{0+}(x, y) * x \neq y \\
\texttt{ls}_{2+}(x, y) &\rightarrow \texttt{ls}_{0+}(x, y) \wedge_{\neg} (x \mapsto y) \\
\texttt{dls}_{1+}(x, y, p, n) &\rightarrow \texttt{dls}_{0+}(x, y, p, n) * x \neq n \\
\texttt{dls}_{2+}(x, y, p, n) &\rightarrow \texttt{dls}_{1+}(x, y, p, n) * x \neq y \\
\texttt{dls}_{3+}(x, y, p, n) &\rightarrow \texttt{dls}_{2+}(x, y, p, n) \wedge_{\neg} (x \mapsto \langle \mathrm{p} : p, \mathrm{n} : y \rangle * y \mapsto \langle \mathrm{p} : x, \mathrm{n} : n \rangle) \\
\texttt{nls}_{1+}(x, y, z) &\rightarrow \texttt{nls}_{0+}(x, y, z) * x \neq y \\
\texttt{nls}_{2+}(x, y, z) &\rightarrow \texttt{nls}_{1+}(x, y, z) \wedge_{\neg} (\exists f'_0.\ x \mapsto \langle \mathrm{t} : y, \mathrm{n} : f'_0 \rangle * \texttt{ls}_{0+}(f'_0, z))
\end{aligned}
$$

The idea behind this translation is to force the list to be non-empty by adding an inequality for the 1+ cases, and to explicitly discard the length one model using guarded negation ($\wedge_{\neg}$) for the 2+ cases. $\texttt{nls}_{2+}(x, y, z)$ describes a nested list where the *top-level list* has at least two allocated nodes. The bound does not place any restriction on the lengths of its sublists. Translating $\texttt{nls}_{2+}(x, y, z)$ requires adding a new logic variable f'_0 that is existentially quantified *under* the negation. If the quantifier were placed before the negation, the meaning would be different, because negation changes existential quantifiers to universal and vice versa.

5.3 Interprocedural Analysis

The analysis of a function call is implemented in `src/func_call.ml` and consists of multiple steps.

5.3.1 Reachability Split

First, the input formula is split into two subformulae by reachability from the function arguments. The reachable subformula is found by first finding all reachable spatial atoms. The algorithm starts with a single variable and finds a spatial atom, where this variable is a source variable. *Source variables* of spatial atoms are variables at the position of x in $\texttt{ls}(x, \_)$, $\texttt{dls}(x, x, \_, \_)$, $\texttt{nls}(x, \_, \_)$, and $x \mapsto \langle \_ \rangle$. This atom is added to the set of reachable atoms, and the algorithm continues recursively with all target variables of the spatial atom. *Target variables* are all variables in spatial atoms that are not source variables. This algorithm is run for each of the argument variables.

After this, the set of all reachable variables is collected from the reachable spatial atoms. The arguments themselves and `nil` are also included in this set. All equivalence classes are then filtered so that only reachable variables remain. Finally, inequalities are filtered so that only those, in which both variables are reachable, remain. Similarly, `freed` atoms are filtered.

The reachable subformula is a merge of the reachable spatial atoms and filtered equivalence classes, inequalities, and `freed` atoms. The unreachable subformula is then composed of all atoms of the original formula not present in the reachable subformula.

This is done to simplify the analysis of the called function as much as possible by not including irrelevant atoms and variables in the formulae. It also increases the efficiency of the summaries cache because the cache input does not contain irrelevant data, which would prevent cache hits with slightly different input formulae.

5.3.2 Anchor Variables

After extracting the reachable subformula, it is necessary to rename variables in the formula from argument names to parameter names. However, simply renaming the variables on function entry and then back on function return would also change the values of variables *after* the call, for example:

```
void set_to_null(Node *x) {
  x = NULL;
  return;
}

a = alloc_node();
set_to_null(a);
a->data = 42;
```

If the argument name `a` were simply renamed to parameter name `x` before analyzing the function `set_to_null`, the original state before calling the function $a \mapsto \texttt{nil}$ would be passed into the function as $x \mapsto \texttt{nil}$. After the analysis of the function body, we would get the state $x = \texttt{nil} * f'_0 \mapsto \texttt{nil}$, which would be simplified to just $x = \texttt{nil}$, and a memory leak would be reported. After returning from the function, the state at the assignment would be $a = \texttt{nil}$, which would falsely report an invalid dereference.

For this reason, *anchor variables* are added to the input formula before the renaming. Anchor variables are treated as regular program variables. Otherwise, simplifications would remove them from the formula. In the case of the program above, the anchor variable A_a will first be added as equal to a, and then a will be renamed to the parameter name x. The resulting state used for the analysis of the function body will then be $x \mapsto \texttt{nil} * x = A_a$.

After the analysis of the function, we will get the formula $f'_0 \mapsto \texttt{nil} * f'_0 = A_a * x = \texttt{nil}$, in which the A_a anchor will be renamed back to a, yielding the original formula $a \mapsto \texttt{nil}$ after subsequent simplification passes.

5.3.3 Function Summaries and Function Analysis

To avoid recomputing the analysis of a function call every time that the function is reached, a cache of *function summaries* is maintained. A function summary is the mapping of an input formula to an output state. Input to the cache is a pair of the called function and the processed input formula after adding anchor variables and renaming arguments to parameters. Caching cannot be done on the original input formula because argument names will differ at different call sites. The conversion of the output state will ensure that a summary created at one call site will be valid at any other call site.

If a summary is not found in the cache, the analysis of the function begins. Unfortunately, the implementation of dataflow analysis in Frama-C does not directly support interprocedural analysis, so we must manually backup the analysis context of the current function and create

a new context for the called function. This context consists of one table for the analysis results and a second table for storing the number of loop iterations in underapproximation mode (see Section 5.7 for more details). The context is represented by the type `function_context` and stored in a global variable.

After creating the new context, the initial state for the first statement in the called function is set to the modified input formula, and the analysis is started on this statement. After the analysis finishes, the result state is read out from the result table, and the original context is restored.

The result state must be processed before it is returned from the transfer function. Each formula of the state is processed separately: first, the anchor variables are renamed back to the original argument names, and then the unreachable subformula is added back to the resulting formula. After this, if the call instruction assigns the call result to a variable, the variable returned by the `return` statement from within the called function is assigned to the call result variable. Frama-C preprocesses all functions to have a single `return` statement with a simple variable as the returned expression.

Finally, all points-to atoms representing pointers to variables on the stack created inside the function are removed, and then all program variables from inside the function are substituted with logic variables, to be removed by the subsequent simplification passes. Formulae processed this way are returned from the transfer function.

5.4 Simplifications

Simplifications are done on the formulae of the state between the analysis of two instructions to reduce the size of formulae and the number of variables in them to speed up solver queries. Most of the simplification passes are implemented in `src/simplification.ml` and called from the `doEdge` function inside `src/analysis.ml`.

Substitution of program variables going out of scope by logic variables is done to enable the abstraction to work. Abstraction can only remove logic variables from the formula, so if variables created within loops were not turned into logic variables at the end of the loop body, it would never have the option to introduce list predicates into the formula. As an example, consider the following code:

```
List *a = allocate_node();
while (rand()) {
  List *b = allocate_node();
  a->next = b;
  b->next = NULL;
}
```

The formula describing the state at the end of the first loop iteration might be $a \mapsto b * b \mapsto \texttt{nil}$. Because the variable b is going out of scope at the end of the loop body, the formula will be simplified to $a \mapsto f_0' * f_0' \mapsto \texttt{nil}$, which will enable the abstraction into $\texttt{ls}_{2+}(a, \texttt{nil})$ before the analysis returns to the start of the loop.

Spatial atoms starting at a logic variable are removed if the logic variable does not appear in any other spatial or equality atom. It can appear in distinct and `freed` atoms because these do not make the logic variable reachable from any program variable in the formula. The removal of a spatial atom is reported as a memory leak, except when removing a list with a length bound of zero. Reporting these would cause false detections of memory leaks. For example,

when unrolling a nested list, an SLS atom with the length bound zero is added, no matter the original sublist length.

Removing logic variables in equivalence classes is done by first filtering out equivalence classes with fewer than two variables. Then, in each equivalence class, a substitution target variable is found according to the following rules in order:

- If the equivalence class contains `nil`, it is used as the substitution target.
- If the equivalence class contains a program variable, it is used.
- Otherwise, a logic variable is used.

All logic variables in each equivalence class are then substituted with its substitution target. After this, all equivalence classes are deduplicated, getting rid of the variables duplicated during substitution. Finally, equivalence classes with fewer than two variables are again removed.

Formula canonicalization is done to convert formulae into a syntactically consistent shape, to increase the efficiency of the query cache. Canonicalization is done in these steps:

- All variables of the formula are collected and sorted, and then explicitly set as the source of spatial atoms. Variables that are not a source of any spatial atom are skipped.
- Variables in equivalence classes and inequalities are sorted.
- Atoms in the formula are sorted.
- All logic variables in the formula are in order renamed to a sequence of names of the form `!0`, `!1`, `!2`, …

This is certainly not an ideal way to canonicalize formulae because even a single difference in variable names between two formulae can affect their final order of atoms, but it proved sufficient for the next step to work.

Generalization of similar formulae is done only when two formulae differ in the length bound of a single spatial atom. This comparison is run for every pair of formulae in the state. If it succeeds, the pair is replaced with the generalized formula. Otherwise, it is left unchanged. The length bound in the generalized formula is the smaller of the original two. If two formulae differ in a single atom and one of the differing atoms is a points-to atom, it is converted to a list atom with the length bound of one. If the generalization fails, this change is not propagated to the original state. This simplification solves the common problem where an allocation loop produces multiple formulae describing different lengths of a list.

```
a = NULL;
while (rand()) {
  alloc_node(&a);
}
```

This analysis might produce the following state after the loop:

$$\bigvee \begin{bmatrix} a = \texttt{nil} \\ a \mapsto \texttt{nil} \\ \texttt{ls}_{2+}(a, \texttt{nil}) \end{bmatrix}$$

This generalization will merge the latter formulae into one:

$$\bigvee \begin{bmatrix} a = \texttt{nil} \\ \texttt{ls}_{1+}(a, \texttt{nil}) \end{bmatrix}$$

This generalization could be extended to turn the resulting state into $\texttt{ls}_{0+}(a, \texttt{nil})$ that would cover all cases, but it would require a more complex method than to just compare a single pair of differing atoms, because the equality of a and `nil` could exist inside a larger equivalence class.

5.5 Abstraction

The abstraction of each list type is done in a separate pass, separately for each formula. In general, the point of abstracting chains of points-to predicates into list predicates is to create invariants for loops. Abstraction is only done when returning from the end of a loop back to the beginning (i.e., from `b = b->next;` to `while (rand())`). Consider the following code:

```
a = alloc_node();
b = a;

while (rand()) {
  b->next = alloc_node();
  b = b->next;
}
```

If there were no abstraction in place, the analysis would generate longer and longer points-to chains forever:

$$
\begin{aligned}
&a \mapsto \texttt{nil} * a = b \\
&a \mapsto b \quad * b \mapsto \texttt{nil} \\
&a \mapsto f'_0 \quad * f'_0 \mapsto b \quad * b \mapsto \texttt{nil} \\
&a \mapsto f'_0 \quad * f'_0 \mapsto f'_1 * f'_1 \mapsto b \quad * b \mapsto \texttt{nil} \\
&\ldots
\end{aligned}
$$

The abstraction will instead turn the third formula into $\texttt{ls}_{2+}(a, b) * b \mapsto \texttt{nil}$, which will abstract over all possible lengths of the points-to chain that would otherwise be generated. This formula (along with the first two generated formulae) will serve as the fixpoint for this loop (a state that is valid in every iteration of the loop). This is because all possible lengths of the points-to chain together make the set of models of the abstracted formula:

$$
\begin{aligned}
&a \mapsto f'_0 * f'_0 \mapsto b \quad * b \mapsto \texttt{nil} && \models \texttt{ls}_{2+}(a, b) * b \mapsto \texttt{nil} \\
&a \mapsto f'_0 * f'_0 \mapsto f'_1 * f'_1 \mapsto b * b \mapsto \texttt{nil} && \models \texttt{ls}_{2+}(a, b) * b \mapsto \texttt{nil} \\
&a \mapsto f'_0 * f'_0 \mapsto f'_1 * f'_1 \mapsto f'_2 * f'_2 \mapsto b * b \mapsto \texttt{nil} && \models \texttt{ls}_{2+}(a, b) * b \mapsto \texttt{nil} \\
&\quad \ldots
\end{aligned}
$$

5.5.1 Singly-Linked Lists

Abstraction of SLSs is done by iterating through all spatial atoms with the appropriate sort and trying to find a second spatial atom starting at the end of the first one. Points-to atoms are again treated as list atoms with a length bound of one. For example, when processing $\texttt{ls}_{0+}(x, y)$, the algorithm would look for a spatial atom starting at y. If found, the following conditions must be met to proceed with the abstraction:

- The middle variable must be a logic variable not occurring in the rest of the formula. Otherwise, joining $\texttt{ls}_{0+}(x, y) * \texttt{ls}_{1+}(y, z)$ into $\texttt{ls}_{1+}(x, z)$ would lose information about the variable y. The occurrence of the middle variable in the formula is checked only in spatial and equivalence atoms, because the variable can only be reached through these.
- The source variable of the first list must be distinct from the target variable of the second list. In the previous example, this would mean ensuring that $x \neq z$. This is checked using the

solver by adding an equality of the two variables into the formula and checking satisfiability. If such formula is satisfiable, the variables are not distinct, and the abstraction cannot be done because the predicate $\texttt{ls}(x, y)$ describes an *acyclic* chain of pointers.

If the conditions are met, the two atoms are replaced with a single list atom of length $\texttt{min}(l_1 + l_2, 2)$ where l_1 and l_2 are the lengths of the original lists. For example: $x \mapsto f'_0 * \texttt{ls}_{2+}(f'_0, \texttt{nil})$ will be abstracted to $\texttt{ls}_{2+}(x, \texttt{nil})$.

5.5.2 Doubly-Linked Lists

The abstraction of DLSs is done similarly, except that the conditions needed for abstraction are slightly different. For abstracting $\texttt{dls}_{2+}(x, f'_0, \texttt{nil}, f'_1) * \texttt{dls}_{2+}(f'_1, y, f'_0, \texttt{nil})$ into $\texttt{dls}_{3+}(x, y, \texttt{nil}, \texttt{nil})$, the following conditions must hold:

- Either the first and the last allocated variable of both atoms must be the same (that signifies the lists have length at most one), or the last allocated variable of the first list (f'_0) and the first allocated variable of the second list (f'_1) must be logic variables unreachable from the rest of the formula, like in the SLS case.
- The `p` field of the second list must point back to the last allocated variable of the first list.
- The `p` and `n` fields of the first allocated variable of the first list and the last allocated variable of the last list, respectively, cannot point back into the list.
- Like in the SLS case, the list must not be cyclic – this is checked using the solver both forward and backward.

The length bound of the joined list will be $\texttt{min}(l_1 + l_2, 3)$, because unlike in SLS, there is a way to translate DLSs with a minimum length of three.

5.5.3 Nested Lists

NLSs are abstracted similarly to SLSs, except that an additional check must be done that the SLS sublists end up in the same variable. SLS abstraction is run first, so we can expect the sublists to be abstracted into a single list atom. The algorithm for joining two NLS atoms differs based on the type of atom:

When joining two points-to atoms, $x \mapsto \langle \text{t} : f'_0, \text{n} : f'_1 \rangle * f'_0 \mapsto \langle \text{t} : y, \text{n} : f'_2 \rangle$, the following options are tried in order:

- If f'_1 is equal to f'_2, the atoms are joined as-is into $\texttt{nls}_{2+}(x, y, f'_1)$.
- If the formula contains $\texttt{ls}(f'_1, z)$ and z is equal to f'_2, the sublist from f'_1 is removed and the atoms are joined into $\texttt{nls}_{2+}(x, y, z)$.
- If the formula contains $\texttt{ls}(f'_1, z_1) * \texttt{ls}(f'_2, z_2)$ and z_1 is equal to z_2, the sublists from f'_1 and f'_2 are removed and the atoms are joined into $\texttt{nls}_{2+}(x, y, z_1)$.

When joining a list atom $\texttt{nls}(x, f'_0, z) * f'_0 \mapsto \langle \text{t} : y, \text{n} : f'_1 \rangle$, these options are tried in order:

- If z is equal to f'_1, the atoms are joined as-is into $\texttt{nls}(x, y, z)$.
- If the formula contains $\texttt{ls}(f'_1, f'_2)$ and f'_2 is equal to z, the sublist from f'_1 is removed and the atoms are joined into $\texttt{nls}(x, y, z)$.

The only way to join two list atoms $\texttt{nls}(x, f'_0, z_1) * \texttt{nls}(f'_0, y, z_2)$ into $\texttt{nls}(x, y, z)$ is when z_1 and z_2 are already equal.

All equivalences are checked syntactically. All variables at the start of the deleted sublists must be logic variables unreachable from the rest of the formula. The reason for checking this many options instead of just looking for sublists everywhere is that an $\texttt{nls}(x, y, z)$

atom already contains sublists leading to z. For example, it is not possible to join $\mathtt{nls}(x, f_0', f_1') * \mathtt{ls}(f_1', z) * f_0' \mapsto \langle \mathrm{t} : y, \mathrm{n} : z \rangle$ into $\mathtt{nls}(x, y, z)$ because that would violate the semantics of the NLS atom, in that all sublists leading into z must be disjoint.

5.6 Deduplication Sorting

Because the deduplication algorithm in the join operation processes formulae in order, it is beneficial to first sort these formulae so that the chances of finding duplicate formulae are as high as possible. Specifically, it is always better to have a higher length bound on the list atoms on the left side of the entailment. For example, the entailment $\mathtt{ls}_{0+}(x, y) \vDash \mathtt{ls}_{2+}(x, y)$ will not succeed, but $\mathtt{ls}_{2+}(x, y) \vDash \mathtt{ls}_{0+}(x, y)$ will.

The sorting is done by defining a comparison function that accepts two formulae and puts them in the correct order. This function first calculates the lowest number of allocations for each of the formulae and then puts the higher of the two on the left side of the entailment. This makes sense because the formula with the higher number of allocations can have a subset of the models of the other formula, but not the other way around. For example, this entailment holds:

$$\mathtt{ls}_{2+}(x, y) \vDash \mathtt{ls}_{1+}(x, y)$$

But this does not:

$$\mathtt{ls}_{1+}(x, y) \vDash \mathtt{ls}_{2+}(x, y)$$

The score for a formula is calculated by adding up all length bounds on its list atoms, while points-to atoms are counted as 1. Additionally, if a formula contains any number of list atoms, its score is decreased by one. This is done to prevent a points-to atom from being on the right side of an entailment:

$$\begin{gathered} x \mapsto y \vDash \mathtt{ls}_{1+}(x, y) \\ 1 > 0 \end{gathered}$$

This method also works for some combinations of points-to and list predicates:

$$\begin{gathered} x \mapsto f_0' * \mathtt{ls}_{0+}(f_0', y) \vDash \mathtt{ls}_{0+}(x, y) \\ 0 > -1 \end{gathered}$$

And for multiple list atoms in one formula:

$$\begin{gathered} \mathtt{ls}_{1+}(x, f_0') * \mathtt{ls}_{2+}(f_0', y) \vDash \mathtt{ls}_{2+}(x, y) \\ 2 > 1 \end{gathered}$$

If this comparison does not decide the order of the entailments, a second score is calculated. The algorithm is the same, except that points-to atoms are not counted at all. In this case, the formula with the lower score is placed on the left side of the entailment. This will order formulae such as these:

$$\begin{aligned} x \mapsto f_0' * \mathtt{ls}_{1+}(f_0', y) &\vDash \mathtt{ls}_{2+}(x, y) \\ 1 &\not> 1 \qquad \text{(first score)} \\ 0 &< 1 \qquad \text{(second score)} \end{aligned}$$

The second scoring does not affect whether the entailment will succeed or not, but it will keep the simpler, more abstracted formula in the deduplicated list instead of the materialized form.

5.7 Configuration

There are a few configuration options to change how the analysis performs certain operations. Here is a list of the relevant ones.

`-sl-edge-abstraction` enables the abstraction of formulae on every edge between two statements, not just at the end of a loop. This can significantly decrease the analysis times when enabled because the abstraction is done earlier, but can lead to less precise results on some programs because of earlier over-approximation.

`-sl-no-edge-deduplication` disables formula deduplication on every edge between two statements. Note that even when this option is enabled, deduplication is done during the join operation.

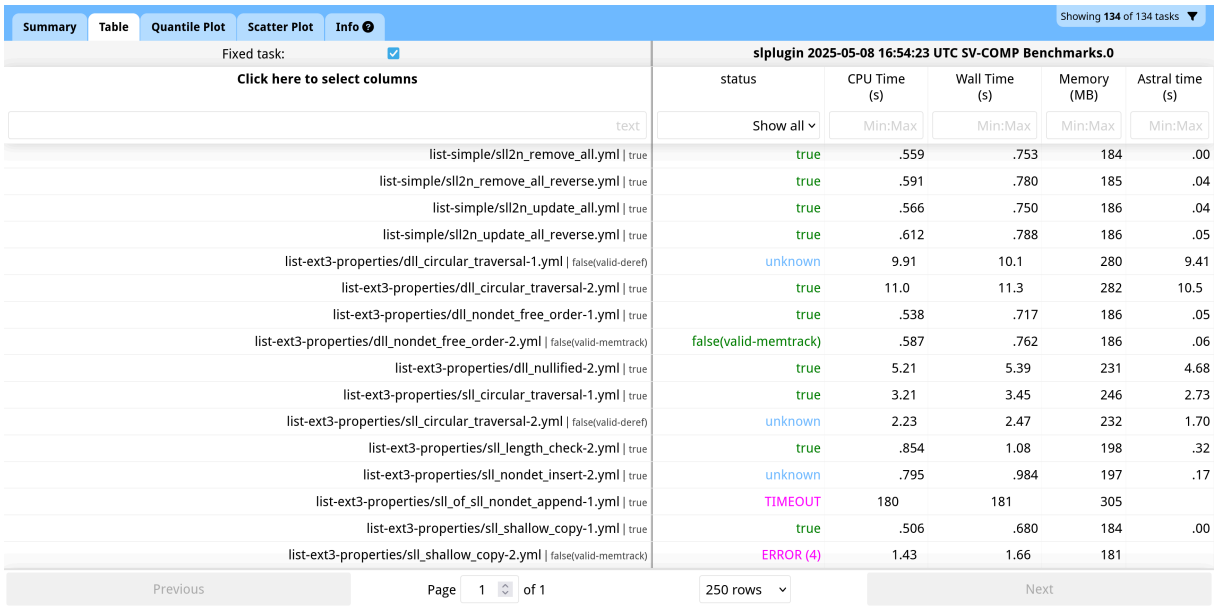
`-sl-no-simple-join` disables the optimization in formula deduplication. When this option is used, the entailments are computed directly using disjunctions of formulae instead of formula by formula.

`-sl-max-loop-cycles <N>` enables an underapproximation mode, where each loop is traversed at most `N` times when reached. When enabled, the analysis no longer proves the correctness of programs, but the output of the analysis can still help find bugs in programs where the normal analysis would get stuck in a loop, for which it cannot find a fixpoint.

6 Results

The KTSN analyzer was evaluated on a small set of crafted programs and on a larger set of SV-COMP benchmarks [20] to compare it with existing tools.

We use the Benchexec framework [21] to run experiments on both sets of benchmarks. Benchexec is a benchmarking framework that automatically evaluates a program on a set of benchmarks while constraining system resources. Benchexec requires a tool definition in the form of a Python module that tells it how to execute the analyzer on a given test program and how to extract its result. This module is provided in `bench/ktsn/ktsn.py`. The benchmarks are then executed using a benchmark run definition, an XML file that defines the set of programs on which to test the analyzer and what resource limits to apply. Benchexec then runs the benchmarks while tracking resource usage. The results of all the benchmarks can then be rendered to an HTML table for viewing or exported to CSV for further processing.



Summary | Table | Quantile Plot | Scatter Plot | Info — Showing 134 of 134 tasks

Fixed task: ☑ — slplugin 2025-05-08 16:54:23 UTC SV-COMP Benchmarks.0

Click here to select columns

	status	CPU Time (s)	Wall Time (s)	Memory (MB)	Astral time (s)
text	Show all	Min:Max	Min:Max	Min:Max	Min:Max
list-simple/sll2n_remove_all.yml | true	true	.559	.753	184	.00
list-simple/sll2n_remove_all_reverse.yml | true	true	.591	.780	185	.04
list-simple/sll2n_update_all.yml | true	true	.566	.750	186	.04
list-simple/sll2n_update_all_reverse.yml | true	true	.612	.788	186	.05
list-ext3-properties/dll_circular_traversal-1.yml | false(valid-deref)	unknown	9.91	10.1	280	9.41
list-ext3-properties/dll_circular_traversal-2.yml | true	true	11.0	11.3	282	10.5
list-ext3-properties/dll_nondet_free_order-1.yml | true	true	.538	.717	186	.05
list-ext3-properties/dll_nondet_free_order-2.yml | false(valid-memtrack)	false(valid-memtrack)	.587	.762	186	.06
list-ext3-properties/dll_nullified-2.yml | true	true	5.21	5.39	231	4.68
list-ext3-properties/sll_circular_traversal-1.yml | true	true	3.21	3.45	246	2.73
list-ext3-properties/sll_circular_traversal-2.yml | false(valid-deref)	unknown	2.23	2.47	232	1.70
list-ext3-properties/sll_length_check-2.yml | true	true	.854	1.08	198	.32
list-ext3-properties/sll_nondet_insert-2.yml | true	unknown	.795	.984	197	.17
list-ext3-properties/sll_of_sll_nondet_append-1.yml | true	TIMEOUT	180	181	305	
list-ext3-properties/sll_shallow_copy-1.yml | true	true	.506	.680	184	.00
list-ext3-properties/sll_shallow_copy-2.yml | false(valid-memtrack)	ERROR (4)	1.43	1.66	181	

Previous — Page 1 of 1 — 250 rows — Next

Figure 3: Benchmark results rendered by Benchexec, green results are correct (`true` for correct programs, `false(property-name)` for bug detections), purple results signify interrupted tests, incorrect results are marked red.

All benchmarks were run on an AMD EPYC 9124 processor with 1500 MB of available memory and three minutes of total run time per benchmark.

The amount of memory was chosen to be as small as possible, such that no tests would run out of memory and that the testing could be parallelized into eight processes on a VM with 8 CPU cores and 16 GB of available memory. This limit has a wide margin since the largest amount of allocated memory during a benchmark was under 600 MB. In the case of maximum test time, the period of three minutes was chosen as the smallest value such that no additional tests would timeout compared to a run with 15 minutes of maximum test time.

The evaluated version of the analyzer is marked with the Git tag `bp_version` in the tool's repository at `https://github.com/pepega007xd/ktsn`.

In the SV-COMP dataset, one of the analyzed properties of programs is called `MemSafety`, which requires the following subproperties to hold:

- `valid-free` – all memory deallocations are valid (no pointers other than those allocated with `malloc` or other allocation functions are passed to the `free` function).

- `valid-deref` – all pointer dereferences are valid.
- `valid-memtrack` – all memory allocations are tracked (no memory is leaked by losing a pointer to it or left allocated after the end of the `main` function).

The benchmarked programs all have an expected result, which can either be a violation of one of these properties, or `true`, signifying that all properties are valid for the program.

6.1 Crafted Benchmarks

The crafted benchmarks verify that all basic parts of the analyzer work as intended. On the evaluated version of the analyzer, all of the crafted benchmarks are passing. Since these programs are purposefully written to test the analyzer, they do not contain any code that requires the external preprocessing described in Section 5.1.3, and it is therefore not enabled for these tests. The test programs are located in `test_programs` and can be divided into these groups.

The first group contains programs that test a specific part of the analysis, which serve as quick checks of specific functionality during development. The only property being tested here is that the analysis completes successfully. This group includes:

- `all_list_types.c` – list and field type recognition, type heuristic,
- `generic_structs.c` – handling of structs not recognized as lists,
- `stack_pointers.c` – handling of pointers to variables on the stack.

The second group tests the analysis of a correct program that allocates a list of nondeterministic size, traverses the list from start to end, and deallocates the list. These benchmarks allow us to compare the run time of an equivalent program for different list types (programs named `*_full.c`). These tests also verify that all basic parts of the analysis (abstraction, simplification) are working as expected. The second group contains:

- `ls_full.c`,
- `ls_full_return.c`,
- `ls_full_single_function.c`,
- `ls_cyclic.c`,
- `dls_full.c`,
- `nls_full.c`.

Program	CPU Time
`ls_full.c`	1.05 s
`dls_full.c`	13.1 s
`nls_full.c`	17.0 s

As you can see in the table above, the analysis of programs using doubly-linked lists and nested lists is roughly an order of magnitude longer than the analysis of singly-linked lists.

The third group tests the detection of different types of bugs. The only property being tested is that the errors are reported as the correct type of error, since, for example, the `valid-free` property is not violated in any of the SV-COMP benchmarks of the `LinkedLists` subset. This group includes:

- `ls_null_deref.c` – detection of a possible null-pointer dereference (`valid-deref` property),
- `ls_use_after_free.c` – detection of an access to a freed allocation (`valid-deref` property),
- `dls_double_free.c` – detection of a double-free (`valid-free` property),
- `nls_memory_leak.c` – detection of a memory leak (`valid-memtrack` property).

The last group tests some other interesting operations on singly-linked lists:

- `ls_merge_lists.c` – program that joins one list on the end of another list,
- `reverse_list.c` – program that reverses the order of a list,
- `postfix_calculator.c` – program that evaluates simple mathematical expressions in Reverse Polish (postfix) notation. The stack used to hold operands is implemented using a linked list.

Program	CPU Time
`ls_merge_lists.c`	7.89 s
`reverse_list.c`	2.36 s
`postfix_calculator.c`	1.01 s

As can be seen from the analysis times, the complexity of the program itself does not affect the run time of the analysis. Even though the calculator is a larger program than most of the other tested programs (147 lines of code), the analysis takes only a second. The complexity of the operations on the linked lists is what dictates the analysis time. In the case of the calculator, the only operations on the list are simple `push` and `pop` operations that do not require the traversal of the list.

6.2 SV-COMP Benchmarks

SV-COMP is a yearly competition for verification tools composed of many categories, each verifying a different property of programs. One of these categories is `MemSafety`, in which analyzers verify the properties described previously. However, the `MemSafety` category includes many kinds of programs, such as those working with arrays, for which our analyzer cannot succeed. Therefore, we restricted the set of benchmarks only to the `LinkedLists` subset of this category.

Since many analyzers do not support data structures of unbounded size, the benchmarks contain a number of programs that work with lists of small, bounded size. To improve the precision of our analysis on these benchmarks, we unroll a fixed number of iterations on loops (see Section 5.1.3).

When trying different settings of the loop unrolling, it became clear that some tests will only finish when run with a higher count of unrolled iterations, and other tests will only finish with a lower count. After some trial and error, we chose to run the analysis for one third of the maximum time with the unrolling level set to 3. If the analysis times out, it is run with the unrolling level of 2 for another third of the time, and for the last third, no unrolling is done at all. This yielded some improvements, but a better solution would be to implement a custom loop unrolling pass that would only unroll loops with a fixed number of iterations.

Of the 134 total benchmarks, this tool correctly analyzes 80 programs. The full results can be seen in the table below, compared to PredatorHP (the best-performing analyzer of SV-COMP 2025 in the `LinkedLists` subcategory) and EVA (static analyzer built into Frama-C).

Table 1: The results of evaluating different analyzers on the SV-COMP dataset

Tests (134 total)	**KTSN**	**PredatorHP**	**EVA**
Correct	80	124	56
Correct (true)	74	96	50
Correct (false)	6	28	6
Incorrect (true)	0	0	6
Incorrect (false)	0	0	48
Timeout	11	10	4
Unknown	43	0	20

In the table below, you can see that our analyzer outperforms all but two other tools in the `LinkedLists` subcategory. Note that a number of tools can only analyze programs without unbounded data structures, as they are not specialized for linked lists but focus on memory safety in general. Also note that the overall winner of SV-COMP 2025, ULTIMATE Atomizer, correctly analyzes only 12 programs of this subcategory.

Table 2: The number of correct results in the `LinkedLists` subcategory for different analyzers

Analyzer	**Correct Results**
PredatorHP	124
CPAchecker	118
nacpa (CPAchecker)	118
KTSN	**80**
symbiotic	79
ESBMC	78
CBMC	77
Bubaak	66
Graves-CPA	64
PeSCo	64
2LS	60
Mopsa	48
DIVINE	42
sv-sanitizers	17
ULTIMATE Automizer	12
ULTIMATE Taipan	8
ULTIMATE Kojak	6
Goblint	1
SVF-SVC	0
Theta	0

The failing benchmarks can be divided into the following categories:

Failure reason	Number of benchmarks
Unsupported language features	25
Timeout	11
Unknown result	18

First, there are benchmarks that use code from the Linux kernel. This includes all benchmarks from the `ddv-machzwd` group and some benchmarks from `memsafety-broom`. These fail immediately since the code contains uses of pointer arithmetic, which is unsupported. Some other tests outside of the kernel code also use unsupported language features, such as the allocation of structures directly on the stack or structure types nested in other structure types, and therefore fail at the preprocessing level.

In the group of benchmarks that timeout, we can find a few problems. Most of these programs use structs that will not be detected as the correct list type. This is the case in `forester-heap/dll-01-2.c`, `list-ext3-properties/sll_of_sll_nondet_append-1.c`, and others. In these programs, the abstraction will never be able to create list predicates because the structures in the heap will be different from what is expected, and the analysis will not terminate. Some programs implement lists that are simply not covered by the supported list types, for example `heap-manipulation/tree-1.c`, which implements a binary tree. A few programs that timeout do not have a fundamental reason why the analysis cannot succeed, such as `memsafety-broom/sll-nested-sll-inline.c`. These will require further optimizations of the analyzer or the SL solver to pass.

The rest of the failed benchmarks are those in which a property violation is detected, but because a nondeterministic condition has been reached during the analysis, an unknown result is reported. This problem is described in detail in Section 5.1.3. However, constant propagation and loop unrolling cannot solve all of these cases, especially not those where relevant non-pointer data is stored in the list nodes themselves. This includes most red-black list benchmarks (such as `forester-heap/dll-rb-cnstr_1-1.c`) or benchmarks where integer indices are used to manipulate list nodes (`list-ext3-properties/sll_nondet_insert-2.c`).

The analysis of most correct results is generally fast, 49 of the 80 correct results are analyzed under a second, and 67 under 10 seconds. Because the total analysis time is split into thirds, in which the tool runs with different settings, you can see a group of results appear after one minute.

Compared to PredatorHP, which reaches 119 of its 124 correct results in under a second, the analysis is overall slower. However, CPAchecker does not produce any results in less than 7 seconds, so compared to this tool, our analysis times are overall better. Note that the results of PredatorHP and CPAchecker are taken directly from the SV-COMP results, which was run on a more powerful hardware than ours, and therefore the relative analysis times might be slightly different.

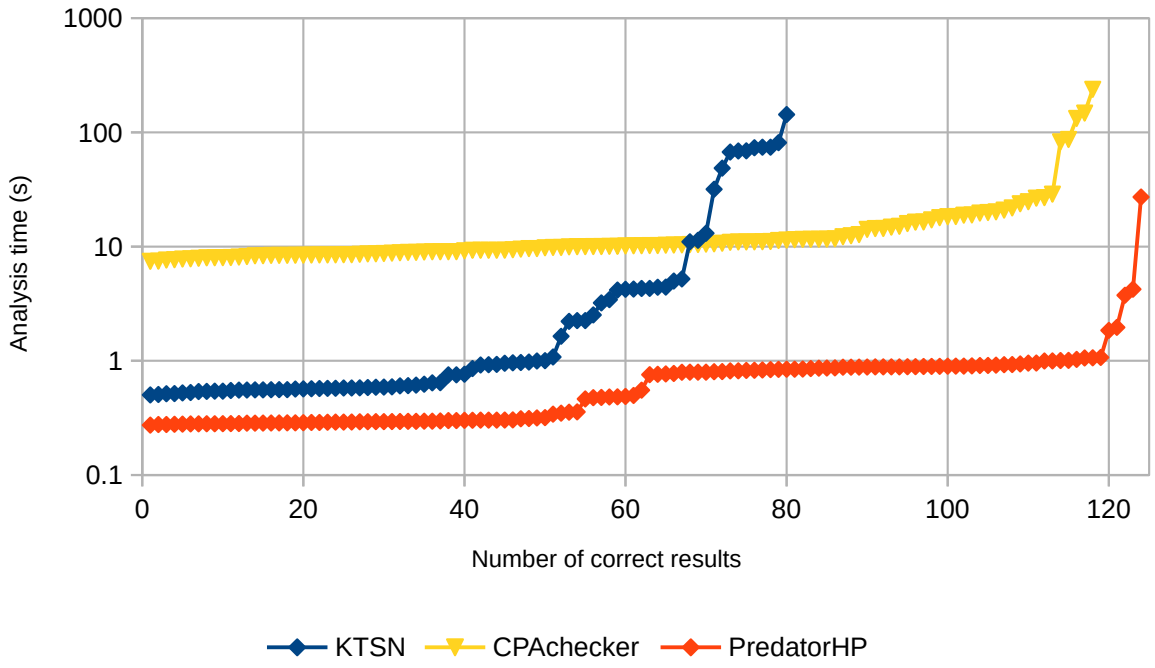


Figure 4: The number of correct results reached by
the analyzers with increasing analysis time (logarithmic axis)

In the next table, we can see how the optimized formula deduplication described in Section 4.2.3 improves the analysis times. With the optimization turned off, the analysis times are consistently higher, but more importantly, some tests do not finish at all. Without the optimization, only 77 benchmarks produce correct results.

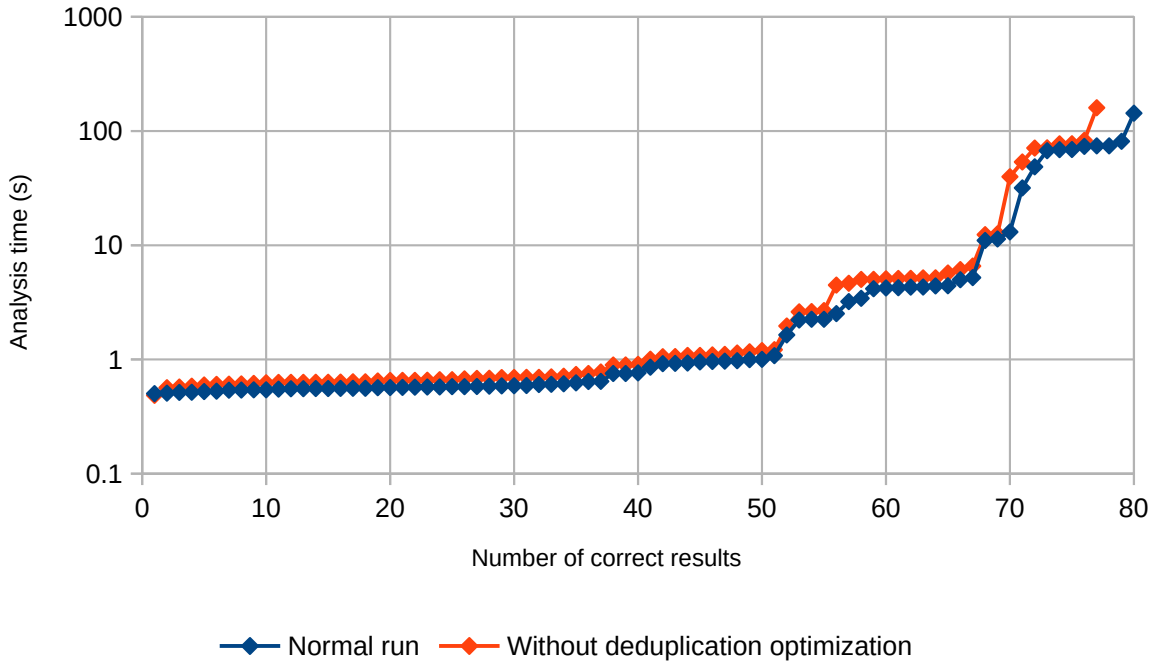


Figure 5: Comparison of a normal benchmark run
with a run where the deduplication optimization is disabled

6.3 Evaluation of SL Queries

Our analyzer is able to track how much time each run spends in the Astral solver. The table below shows the three fastest and slowest SV-COMP benchmarks. Because the analyzer is killed during a timeout, it does not have a chance to print out the Astral time. Therefore, this table only lists benchmarks that were completed during the first minute.

Table 3: Examples of the fastest and slowest benchmarks and the time spent in the solver versus in the analyzer itself

Benchmark	**Total time (s)**	**Astral time (s)**	**Analyzer time (s)**	**Number of queries**
`sll2c_append_unequal.c`	0.50	0.00	0.50	2
`sll_shallow_copy-1.c`	0.51	0.00	0.51	2
`dll2c_prepend_equal.c`	0.51	0.02	0.49	4
	...			
`dll-rb-cnstr_1-2.c`	16.29	15.64	0.65	141
`sll-sorted-2.c`	31.79	31.65	0.14	1472
`dll-simple-white-blue-2.c`	48.61	48.13	0.48	816

As you can see from the table, the time spent in the analyzer itself is marginal compared to the total run time. This includes the initialization of the framework, preprocessing, and all work done during the dataflow analysis. Both the average and the median time spent in the analyzer are approximately 0.53 seconds. This clearly shows that to achieve faster analysis, it is pointless to optimize the code of the analyzer itself. The only thing that can meaningfully improve the run times is a better usage of the solver through simplifications of formulae.

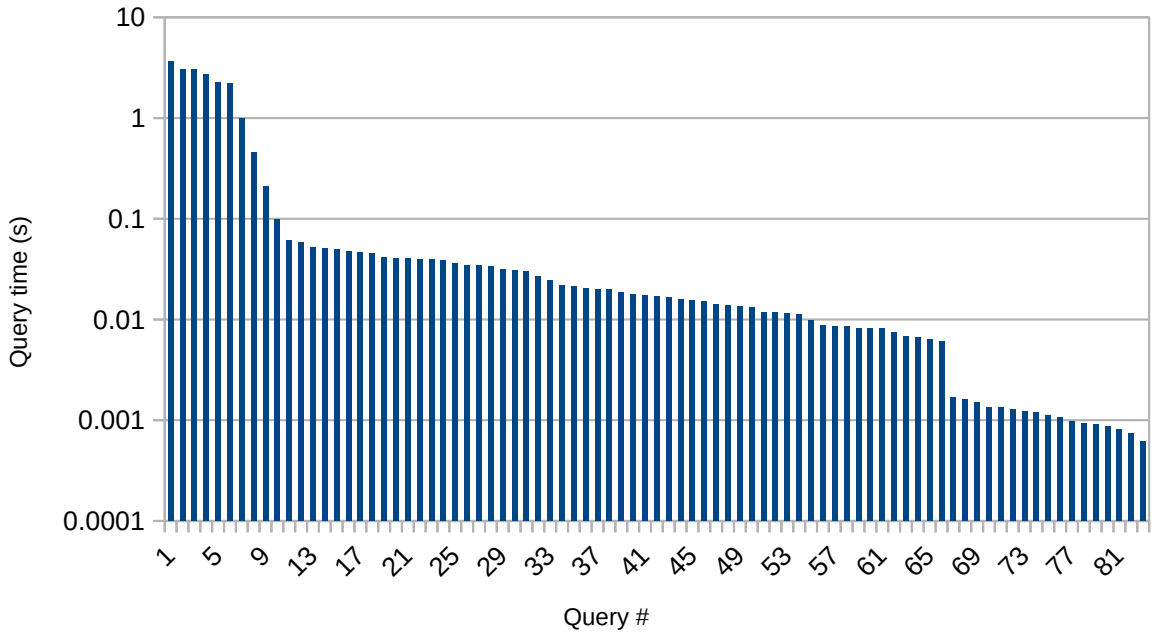


Figure 6: The times of evaluating queries when analyzing `sll-shared-sll-after.c` (logarithmic time axis)

There is usually a small number of queries that take up most of the time spent in the solver. For example, during the analysis of `memsafety-broom/sll-shared-sll-after.c`, the top eight slowest queries take 18.2 seconds of the total 19.9 seconds spent in the solver.

These queries are usually entailments, especially those with logic variables on the right side, as these must be existentially quantified. The longest query in this example (3.60 seconds) is the following entailment:

$$\texttt{nil} = a * b \neq \texttt{nil} * A_b \neq \texttt{nil} * \texttt{freed}(b) * \texttt{freed}(A_b)$$
$$\models$$
$$a \neq \texttt{nil} * A_a \neq \texttt{nil} * \texttt{freed}(a) * \texttt{freed}(A_a) * f'_0 \rightarrow \texttt{nil} * \texttt{ls}_{0+}(b, f'_0)$$

The other slow queries are similar to this one in that they are entailments with list predicates and logic variables on the right side.

7 Conclusion

This thesis introduced the KTSN static analyzer focused on verifying the memory safety of C programs. The tool uses dataflow analysis with separation logic formulae as the value domain. The analyzer implements abstraction using inductive predicates, which allows it to analyze list structures of unbounded size.

The analyzer was tested on a set of crafted programs which show it can prove the correctness of programs that create unbounded lists of all supported types. These benchmarks also verify that the analyzer is able to correctly detect use-after-free errors, null-pointer dereferences, double-free errors, and memory leaks.

Further benchmarking was done on the `LinkedLists` subset of the SV-COMP benchmarks. Of the 134 total benchmarks, KTSN can correctly analyze 80 programs. From the 18 analyzers that competed in SV-COMP 2025, only two others achieve better results than ours.

The use of a dedicated SL solver means that our analysis will benefit from future improvements to the Astral solver and the SMT backends it uses, unlike other analyzers that use custom formalisms for which there are no dedicated solvers. Compared to other tools, another advantage of our SL-based method is that it can be easily extended to other inductive predicates when implemented by the solver.

The main drawback of our method, preventing us from analyzing more programs, is the lack of support for pointer arithmetic. Another disadvantage compared to other tools is the lack of analysis of numeric values, which makes the analysis of integer conditions impossible. Because of this, most programs containing bugs are reported as unknown results instead of detected errors, because the analyzer cannot guarantee that the bug is real, and not just a false-positive caused by an imprecise analysis of a condition.

The analysis time differs based on the type of list involved and the complexity of the operations done on the list. For programs that work with singly-linked lists, the analysis time is usually around a second, often under a second. For doubly-linked lists and nested lists, the analysis can take seconds to tens of seconds. Most of the analysis time is spent in the Astral solver deciding formulae, whereas the time spent in the analyzer itself is roughly independent of the analyzed program, around half a second. Most queries to the solver are evaluated in less than 100 milliseconds, but each analysis usually contains just a small number of queries that take up most of the total analysis time. These queries are typically entailments with inductive predicates and existentially quantified logic variables.

7.1 Future Work

The most promising direction to improve the precision of the analysis is to add a numeric value analysis to the tool. One approach might be to implement a simple numeric value analysis manually. Because Astral decides SL formulae by translating them into SMT, adding SMT terms with integer variables into SL formulae is possible. Another, more interesting way to get a numeric value analysis would be to integrate our analysis method into the EVA analyzer. EVA would provide the values of numeric variables to our tool to better analyze integer conditions, and our analyzer would provide more precise information about pointer variables to EVA. Moreover, the tools are based on similar analysis methods, and both use the same framework, which would simplify the integration.

Another possible improvement is switching from static list type detection based on C types to dynamic detection during the analysis based on pointer structures that form in the heap. This would eliminate timeouts caused by not applying the correct abstraction to the formulae.

Other possible improvements include extending the generalization and joining of similar formulae beyond comparing singular atoms, or using the `freed` predicate to signify unallocated memory locations. These improvements could help reduce the amount of formulae in each state during analysis and decrease the time needed for deciding SL formulae.

Bibliography

[1] C. Cimpanu, "Chrome: 70 % of all security bugs are memory safety issues." May 2020.

[2] C. Cimpanu, "Microsoft: 70 percent of all security bugs are memory safety issues." Feb. 2019.

[3] J. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[4] S. Ishtiaq and P. O'Hearn, "Separation and Information Hiding," in *Proc. of POPL'01*, ACM, 2001.

[5] T. Dacík, A. Rogalewicz, T. Vojnar, and F. Zuleger, "Deciding Boolean Separation Logic via Small Models," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., Cham: Springer Nature Switzerland, 2024, pp. 188–206.

[6] J. Signoles, P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, and B. Yakobowski, "Frama-c: a Software Analysis Perspective," 2012, p. . doi: 10.1007/s00165-014-0326-7.

[7] *Guide to Software Verification with Frama-C*. Springer International Publishing, 2024. doi: 10.1007/978-3-031-55608-1.

[8] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, R. N. Horspool, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 213–228.

[9] L. Correnson, "Ivette: A Modern GUI for Frama-C," 2023, pp. 116–131. doi: 10.1007/978-3-031-26236-4_10.

[10] K. Dudka, P. Peringer, and T. Vojnar, "Byte-Precise Verification of Low-Level List Manipulation," in *In Proc. of SAS*, in LCNS, vol. 7935. 2013, pp. 215–237. doi: 10.1007/978-3-642-38856-9_13.

[11] D. Beyer and M. Lingsch-Rosenfeld, "CPAchecker 4.0 as Witness Validator," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Gurfinkel and M. Heule, Eds., Cham: Springer Nature Switzerland, 2025, pp. 192–198.

[12] D. Beyer and M. E. Keremoglu, "CPACHECKER: a tool for configurable software verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, in CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 184–190.

[13] D. Bühler, "EVA, an Evolved Value Analysis for Frama-C: structuring an abstract interpreter through value and state abstractions," 2017. [Online]. Available: http://www.theses.fr/2017REN1S016

[14] C. Calcagno and D. Distefano, "Infer: An Automatic Program Verifier for Memory Safety of C Programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[15] L. Holík, P. Peringer, A. Rogalewicz, V. Šoková, T. Vojnar, and F. Zuleger, "Low-Level Bi-Abduction," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, K. Ali and J. Vitek, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 222. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 1–30. doi: 10.4230/LIPIcs.ECOOP.2022.19.

[16] D. Distefano, P. W. O'Hearn, and H. Yang, "A Local Shape Analysis Based on Separation Logic," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Hermanns and J. Palsberg, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 287–302.

[17] J. Berdine *et al.*, "Shape Analysis for Composite Data Structures," in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 178–192.

[18] "Semantic constant folding." Accessed: May 01, 2025. [Online]. Available: https://www.frama-c.com/fc-plugins/semantic-constant-folding.html

[19] D. Bühler, P. Cuoq, and B. Yakobowski, "Eva – The Evolved Value Analysis plug-in." Accessed: Apr. 25, 2025. [Online]. Available: http://frama-c.com/download/frama-c-eva-manual.pdf

[20] "SV-Benchmarks: Collection of Verification Tasks." Accessed: May 05, 2025. [Online]. Available: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[21] D. Beyer, S. Löwe, and P. Wendler, "Reliable Benchmarking: Requirements and Solutions," *International Journal on Software Tools for Technology Transfer*, vol. 21, 2017.