





2LS: Arrays and Loop Unwinding

(Competition Contribution)

Viktor Malík^{3*}, František Nečas³, Peter Schrammel^{1,2},
and Tomáš Vojnar³

¹Diffblue Ltd., Oxford, UK

²University of Sussex, Sussex, UK

³Brno University of Technology, FIT, Brno, Czech Republic **

imalik@fit.vut.cz

Abstract 2LS is a C program analyser built upon the CPROVER infrastructure that can verify and refute program assertions, memory safety, and termination. Until now, one of the main drawbacks of 2LS was its inability to verify most programs with arrays. This paper introduces a new abstract domain in 2LS for reasoning about the contents of arrays. In addition, we introduce an improved approach to loop unwinding, a crucial component of the 2LS' verification algorithm, which particularly enables finding proofs and counterexamples for programs working with dynamic memory.

1 Overview

2LS is a static analysis and verification tool for sequential C programs. At its core, it uses the $kIkI$ algorithm (k -invariants and k -induction) [2], which integrates bounded model checking, k -induction, and abstract interpretation into a single, scalable framework. $kIkI$ relies on incremental SAT solving in order to find proofs and refutations of assertions, as well as to perform (non)termination analysis [3].

One of the core mechanisms of $kIkI$ is incremental loop unwinding. However, the original unwinding approach that 2LS used was not compatible with the memory model developed in [6]. Hence, in the first part of this paper, we introduce a new approach to loop unwinding [9] that supports programs manipulating dynamic memory and hence allows 2LS to verify programs that could not be handled before.

The abstract interpretation part of $kIkI$ features multiple abstract domains for reasoning about various data structures in programs. In particular, the competition version of 2LS uses the interval domain for numerical values and our custom heap domain for describing the shape of the heap. A common data structure that 2LS could not handle in the past are arrays. Therefore, in the second part of this paper, we introduce a new array abstract domain capable of reasoning about the content of arrays.

Architecture. The architecture of 2LS has been described in previous competition contributions [10,7,8]. In brief, 2LS is built upon the CPROVER infrastructure [4] and thus uses *GOTO programs* as the internal program representation. The analysed program is first translated into a single static assignment (SSA) form. Then, inductive invariants in various abstract domains are computed for the program's loops. Last, the SSA form and the invariants are bit-blasted into a propositional formula and given to a SAT solver which is used to reason about the program's properties.

* Jury member

** The Czech authors were supported by the Czech Science Foundation project 23-06506S, the FIT BUT project FIT-S-23-8151, and the Horizon Europe project CHESS (id 101087529).

Software Project. 2LS is implemented in C++ and it is maintained by Peter Schrammel and Viktor Malík with contributions by the community. The competition version uses Glucose 4.0 as its back-end SAT solver. 2LS competes in all C categories except Concurrency. See the previous competition report [8] for details on executing 2LS.

2 Loop Unwinding of Heap-Manipulating Programs

Whenever the *kIkI* algorithm is not able to verify or refute the program’s properties for the given unwinding level, it incrementally unwinds the loops in order to compute a stronger invariant or to explore additional reachable program states [2]. 2LS’ original unwinder unrolls the loops directly at the level of the program’s SSA form. However, this approach is not compatible with the encoding of pointer operations that 2LS uses [6]. Hence, for this year’s competition version of 2LS, we introduce a new approach to loop unwinding which overcomes these limitations and allows to verify heap-manipulating programs using *k*-induction and BMC.

Memory model in 2LS. Each call of `malloc` is replaced by a finite number of so-called *abstract dynamic objects* that over-approximate the (possibly unbounded) set of concrete dynamic objects allocated by that call. Subsequently, the conversion of pointer-dereferencing operations to the SSA form is based on a static *points-to* analysis which computes for each pointer *p* the set of memory objects that *p* can be dereferenced into. Reads and writes to memory through *p* are then encoded using a case-split of objects which *p* can point to in the program location of the given memory operation [6].

The *points-to* analysis is performed on the *GOTO program* (control-flow graph) prior to generating the SSA form. This approach poses a problem for the original unwinder when dealing with allocations inside loops. Each new unwinding of a loop may introduce a new call to `malloc`, effectively introducing new abstract dynamic objects. Such additions invalidate the previously computed *points-to* analysis since pointers may now also point to the new objects and, thus, operations via pointers must be re-encoded.

Unwinding in the GOTO programs. Our new approach to loop unwinding unrolls the loops in the *GOTO program* representation instead of the SSA form. This allows us to update the set of abstract dynamic objects in the program as well as to compute the *points-to* analysis anew based on the newly introduced objects [9]. In order to facilitate verification in 2LS, there are multiple transformations that need to be done after the loops of the *GOTO program* are unwound. First, the *k*-induction algorithm of 2LS requires a special unwinding approach. Many state-of-the-art unwinders, including the unwinder from CPROVER that we use, copy the loop body and place it before the original loop (i.e., the unwound loop bodies are outside the loop). On the contrary, 2LS requires all of the unwindings to be included in a single loop, i.e., the backwards edge of the not-yet-unwound part must go to the beginning of the topmost unwinding (instead of going to the top of the not-yet-unwound part) [2]. Hence, we must appropriately reconnect the backwards edges to fulfil this requirement and make our approach usable with the current algorithms of 2LS. Second, assertions inside the unwound loop bodies may be assumed to hold as they were verified in the previous iteration of the *kIkI* algorithm. Hence, 2LS converts such assertions into assumptions. We reflect this approach inside our new unwinding algorithm, cf. [9] for details.

Combining the two approaches. The proposed approach, while being sound when handling dynamic memory, introduces a noticeable performance degradation. Unwinding of loops in the *GOTO* program changes a great part of the generated SSA form which decreases the benefits of incremental SAT solving. To overcome this issue, we only enable the new unwinder when necessary, i.e., when dynamic memory is used in the analysed program. In addition, in our future work, we plan to improve our new unwinder to fully leverage incremental solving.

3 Array Domain

The core algorithm of 2LS, *kIkI*, uses abstract interpretation to infer k -inductive invariants in various abstract domains. The computed invariants are used to verify or refute the program's properties. Since the verification approach of 2LS is based on translating the program into a first-order formula to reason about its properties, the abstract domains in 2LS are required to have the form of a *template*—a parametrised, quantifier-free, first-order formula describing a relevant program property. 2LS already supports a handful of domains, such as the interval domain [2], a shape domain [6], or ranking domains [3] for termination analysis, however, a domain for describing the content of arrays has been missing, which limited usability of 2LS on programs manipulating array structures. In this section, we propose such a domain.

In the literature, there exists a number of works on abstract domains for arrays. To exploit the 2LS' seamless combination of abstract domains, we found that perhaps the most suitable approach to draw inspiration from is [5], where each array is split into several parts, called *segments*, and a separate invariant is computed for every segment. The segment invariant can be computed in any domain supported by 2LS, usually selected based on the data type of the array elements (e.g., the interval domain for numerical values or the shape domain for pointers). In the rest of this section, we describe different aspects of our proposed domain. In all of the below parts, we assume that we compute a loop invariant of an array a . We use N_a to denote the number of elements of a .

Array Segmentation. First, let us assume that we know the set of array indices, so-called *segment borders*, for an array a which we denote B_a (see below on the way this set is obtained). When splitting a into segments, we distinguish two situations:

1. Indices from B_a cannot be totally ordered. In such a case, we create multiple segmentations, one for each $b \in B_a$:

$$\{0\} S_1^b \{b\} S_2^b \{b+1\} S_3^b \{N_a\}. \quad (1)$$

2. Indices from B_a can be totally ordered s.t. $b_1 \leq \dots \leq b_n$. In such a case, we create a single segmentation for the entire a :

$$\{0\} S_1 \{b_1\} S_2 \{b_1+1\} S_3 \{b_2\} \dots \{b_n\} S_{2n} \{b_n+1\} S_{2n+1} \{N_a\}. \quad (2)$$

A single array segment S denoted $\{b_l\} S \{b_u\}$ represents an abstraction of the elements of a between the indices b_l (inclusive) and b_u (exclusive). For each S , we define two special variables: (1) the *segment element variable* $elem^S$ being an abstraction of the array elements contained in S and (2) the *segment index variable* idx^S being an abstraction of the indices of the array elements contained in S .

Array Template. Having the set of program arrays Arr and the set of segments S^a for each $a \in Arr$, we define the array domain template as:

$$\mathcal{T}^A \equiv \bigwedge_{a \in Arr} \bigwedge_{S \in S^a} \left(G^S \Rightarrow \mathcal{T}^{in}(elem^S) \right) \quad (3)$$

where \mathcal{T}^{in} is the inner domain template (over the inner elements of S abstracted by $elem^S$) and G^S is the conjunction of guards associated with the segment S . The purpose of G^S is to make sure that the inner invariant is limited to the elements of the given segment $\{b_l\} S \{b_u\}$. In particular, G^S is a conjunction of several guards:

$$b_l \leq idx^S < b_u \wedge 0 \leq idx^S < N_a \wedge elem^S = a[idx^S] \quad (4)$$

where the first conjunct ensures that the segment index variable stays between the segment borders, the second conjunct makes sure that the segment index variable stays between the array borders (since segment borders are generic expressions, they may lie outside of the array), and the last conjunct binds the segment element variable to the segment index variable. Using the above template, 2LS is able to compute a different invariant for each segment. For example, for a typical array iteration loop, this would allow 2LS to infer a different invariant for the part of the array that has already been traversed than for the part of the array that is still to be visited.

Computing Array Segment Borders. Since 2LS requires the template formula to be fixed at the beginning of the analysis, the set of segments must be pre-computed. The main idea of our approach is that the segment borders should be closely related to the expressions that are used to access array elements in the analysed program. Therefore, we perform a static *array index analysis* which collects the set of all expressions occurring as array access indices (i.e., inside the square bracket operators). Once the analysis is complete, for each array a , we determine the set of its segment borders by taking the set of all index expressions used to write into a in the corresponding loop.

4 Strengths and Weaknesses

For general strengths and weaknesses of 2LS, we refer to the previous competition contribution [8]. The two major improvements described in the previous sections, increase the number of programs correctly verified by this year's version of 2LS. The new loop unwinding approach allows us to use the BMC part of the *kIKI* algorithm for programs manipulating dynamic memory, which particularly enables us to find counter-examples occurring in higher loop iterations, as well as verify such programs for which the initially computed invariant is not sufficiently strong and the loops can be unwound completely. This is the most notable in the heap-related categories (*MemSafety-Heap*, *MemSafety-LinkedLists*, and *ReachSafety-Heap*) where the number of the *correct true* and the *correct false* results increased from 110 to 177 and from 51 to 82, respectively. The new array domain allowed us to score points in array-related categories, which was

not possible before (e.g., 2LS correctly solved 17 tasks in *ReachSafety-Arrays* compared to 2 from the previous years, which 2LS managed by chance)¹.

Still, there remains a number of limitations. The array domain is rather simple and cannot verify many array-manipulating programs. In addition, as we described earlier, the new unwinder cannot make use of incremental SAT solving efficiently.

5 Data-Availability Statement

2LS is publicly available from <https://www.github.com/diffblue/2ls>, under a BSD-style license. The competition version is based on version 0.9.6 and the archive used in the competition is available from <https://doi.org/10.5281/zenodo.7467706> or from the collection of all verifiers and validators participating in SV-COMP 2023 [1].

References

1. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). <https://doi.org/10.5281/zenodo.7627829>
2. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety Verification and Refutation by k -Invariants and k -Induction. In: Proc. of SAS'15. LNCS, vol. 9291. Springer (2015)
3. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-Precise Procedure-Modular Termination Proofs. *TOPLAS* **40** (2017)
4. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Proc. of TACAS'04. LNCS, vol. 2988. Springer (2004)
5. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: Proceedings of the 38th. p. 105–118. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/1926385.1926399>
6. Malík, V., Hruška, M., Schrammel, P., Vojnar, T.: Template-Based Verification of Heap-Manipulating Programs. In: Proc. of FMCAD'18. IEEE (2018)
7. Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T., Wahlang, J.: 2LS: Memory Safety and Non-termination (Competition Contrib.). In: Proc. of TACAS'18. Springer (2018)
8. Malík, V., Schrammel, P., Vojnar, T.: 2ls: Heap analysis and memory safety. In: Proc. of TACAS'20. pp. 368–372. Springer International Publishing (2020)
9. Nečas, F.: Program Loop Unwinding in the 2LS Framework. Bachelor's thesis, Brno University of Technology (2022), <https://www.fit.vut.cz/study/thesis/24719/>
10. Schrammel, P., Kroening, D.: 2LS for Program Analysis (Competition Contribution). In: Proc. of TACAS'16. LNCS, vol. 9636. Springer (2016)

¹ A number of tasks was last-minute disqualified from SV-COMP 2023 due to past-deadline changes which were often related to the tasks being added to new categories (e.g., *NoOverflows*) rather than actual modifications of the tasks or their verdicts. Hence, we present results from the entire benchmark instead of the (limited) competition benchmark set as those results are more representative and can be better compared to the previous year's results.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

