Shape Analysis Using Separation Logic

Tomáš Brablec

Abstract

This work describes the implementation of a static analysis of C programs, that tries to find bugs related to dynamic allocation of memory. The analysis is implemented using the Frama-C framework, and it is done by dataflow analysis using a solver for separation logic. This work also discusses possible improvements to this prototype.

1 Introduction

The way dynamic memory allocation is handled is one of the key aspects that determines the runtime performance of a language. The usual choice for high-level languages is garbage collection, which tracks allocations at runtime, and frees memory when an algorithm determines it is no longer reachable. This has a performance impact that can be unacceptable in certain applications, such as embedded systems. Other languages, notably C and C++, let the user's code manage memory manually. While being the most performant option, this allows for a range of errors resulting from incorrect handling of allocated memory. For example, null pointer dereferences are a common mistake, where a fallible function indicates an error by returning a `NULL` pointer, and the programmer forgets to check for it. Use-after-free is a situation where a pointer to a freed allocation is used to read or write data to memory, resulting in undefined behavior. The opposite situation, where the user does not free memory when it is no longer needed, is called a memory leak. Another common error is double free, where a pointer to already freed allocation is passed to the `free` function again, resulting again in undefined behavior.

To mitigate these problems, a number of methods has been developed. One option is the instrumentation of program binaries with extra code that is able to detect invalid memory accesses at runtime, and safely stop the program. An example of this is Valgrind [1], or LLVM's AddressSanitizer [2]. While being efficient at mitigating the security risks associated with memory errors, these tools come with a performance penalty, and obviously cannot be used to check programs ahead-of-time. Some languages are built to prove program's memory safety statically during compilation; this is achieved by placing additional restrictions on the code, as for example Rust does with its ownership and borrowing rules. While being a reasonably safe option, this makes implementing some programs difficult, or requires writing unsafe code. Static analysis tools exist for other languages, including C, which will be the topic of this work. For example, Facebook's static analyzer Infer can perform analyses of memory allocations.

2 Frama-C framework

Frama-C [3] is a framework for building analysis tools of C99 source code. Frama-C itself is written in the OCaml programming language. Unlike other tools, which focus in finding common bugs using heuristics, Frama-C specializes on verification tools, which guarantee that after successful completion, the program is correct. 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 part is the *C Intermediate Language* (CIL) [4], which the Frama-C kernel constructs for use within plugins, as well as an API to effectively work with it. CIL has the form of an abstract syntax tree of the input source code, with extra information added to it. 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 makes the code more suitable for static analysis. For example, all `return` statements in a function are replaced with `goto` statements to a single `return` at the end of the function. A complete description of CIL can be found in module `Frama_c_kernel.Cil_types` of the Frama-C API documentation [5].

One of the functions of Frama-C is to help with the development of custom analyses. This is done using its plugin system, where a separate plugin binary is built and linked dynamically to the Frama-C runtime. Frama-C then handles command-line argument parsing, reporting results, storing intermediate states of analyses, and exposes APIs to the plugin for access to input's CIL.

2.1 Dataflow analysis

Besides a general framework for crafting analyses, Frama-C provides generic implementations of common algorithms used for static analysis. One of these is dataflow analysis, implemented in module `Frama_c_kernel.Dataflow2`. Dataflow analysis works by assigning an initial value to each node of a Control Flow Graph (CFG), and then systematically updating values of nodes using a transfer function, following the edges between them. When a node is reached for the second time, 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. Typically, a CFG is a structure where a node represents a basic block – a sequence of instructions that will be executed in order, with no loops or conditionals. Edges then represent the control flow of a program - branching and loops.

However, Frama-C implements dataflow analysis in a slightly different way. In dataflow analysis as it is implemented in Frama-C, instructions (`Cil_types.instr`) have a separate transfer function from statements (`Cil_types.stmt`). Instruction is a kind of statement that does not change the control flow of the current function, such as variable definition, assignment, or function call. To implement a dataflow analysis, the user needs to provide the type of data that will be stored and updated for each node of the CFG, and the implementation of the following functions. Before running the analysis, the data for the first statement must be set manually.

- `computeFirstPredecessor` – this function is called when the analysis reaches a statement that has no data associated with it. The input for this instruction is the result of previous statement's transfer function, and the statement itself. The result of this function is the initial state for the given statement, which will be stored. Note that because you set the initial state for the first statement manually, this function is not called for the first statement.
- `doStmt` – this function is used by the transfer function for statements – it gets the statement itself, and data provided by previous statement's transfer function, and it decides what to do. One option is not to continue the analysis of this statement, another option (default) is to continue the analysis of the inside of this statement, and the third is to also continue, but with modified data. Note that the full transfer function for compound statements (if, loop, …) is implemented by the dataflow module itself, the user is not supposed to call `doInstr` inside `doStmt`. Also note that the result of the full transfer function – the new computed data for each statement, is not stored. It is only sent to the next statement's transfer function.
- `doInstr` – this is the transfer function for instructions, it receives data from the englobing statement, and generates new data that will be used by the transfer function of the englobing statement. Note that the result of `doInstr` is also not stored, only sent to the transfer function that called `doInstr` internally.

- `combinePredecessors` – this is the join operation. It is called when the analysis computes a new state for a node that already has some data associated with it. This includes situations, where the analysis goes through two branches of an `if` statement, and then joins data from both branches on the statement immediately following the conditional block. Input for this function is the statement itself, an old state – the data currently stored for the statement, and a new state – the data that was just computed by the transfer function. The returned value is then stored as the data for this statement.
- `doGuard` – this function is called when an `if` statement is reached. It receives the result of the transfer function for the `if` statement, and the condition expression, and generates two states, each to be used in one of the branches.
- `doEdge` – called between analyzing two statements. The function receives both statements, and the result of the first statement's transfer function. The function can modify this data, and the modified version will be passed into the second statement's transfer function.

Note that this is not the full API required for implementing dataflow analysis in `Dataflow2` module, but the other functions, such as pretty-printers for stored data, are not important to the actual analysis, and are therefore omitted.

3 Separation logic

A common technique in program analysis is to generate logical formulae describing the code, and then to use a solver to prove the correctness of the program. However, earlier analyses of programs with dynamic memory have faced a problem with globality – an abstract state of the program would contain the description of the whole heap in such a way, that an update of a single value would require updating the whole program state. Separation logic (SL) aims to solve this inefficiency by describing the heap in a way that allows for local updates of the heap. In this work, I will describe the syntax and semantics of SL [6], for which Tomáš Dacík implemented a solver called Astral [7]. Testing the solver's performance on formulae generated in shape analysis is a secondary goal of this work.

3.1 Syntax

Let $x, y \in \mathbf{Var}$, where $\mathbf{Var}$ is an infinite set of variables, with a special variable $\mathrm{nil} \in \mathbf{Var}$. The syntax of SL is defined by the following grammar.

$$
\begin{aligned}
\varphi &::= x = y \mid x \neq y && \text{(pure atoms)} \\
&\quad \mid x \mapsto y \mid \mathrm{ls}(x, y) && \text{(spatial atoms)} \\
\varphi &::= \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \wedge_{\neg} \varphi \mid \neg\varphi && \text{(boolean connectives)} \\
\varphi &::= \varphi * \varphi \mid \varphi \mathrel{-\!\!\circledast} \varphi && \text{(spatial connectives)}
\end{aligned}
$$

The atomic formulae $x = y$ and $x \neq y$ simply describe equality and inequality of two variables, $x \mapsto y$ corresponds to a pointer from x to y. The atom $\mathrm{ls}(x, y)$ describes a sequence of pointers from x to y, or in other words, an acyclic linked list. The boolean connectives have their usual meaning, the spatial connective $\varphi * \varphi$, called separating conjunction, says basically that the heap can be split into two disjoint parts, each satisfying one of the formulae. The septraction operator $\mathrel{-\!\!\circledast}$ has roughly the following meaning: formula $\varphi_1 \mathrel{-\!\!\circledast} \varphi_2$ is satisfied for all heaps, for which exists another heap satisfying φ_1, which can be merged with our heap, together satisfying φ_2.

3.2 Semantics

The models of SL formulae are so-called stack-heap models. Let **Loc** be an ordered set of memory locations, then stack s is a partial function from **Var** to **Loc**, and heap h is a partial function from **Loc** to **Loc**, where $h(s(\text{nil})) = \perp$. The symbol $\perp$ says that the function h is not defined for this input, which corresponds to an invalid memory access in the analyzed program. Stack-heap model is then simply a pair (s, h). The semantics of equality and inequality atoms are satisfied on all models, where $s(x) = s(y)$, or $s(s) \neq s(y)$ respectively, and the heap is empty. Points-to atom $x \mapsto y$ is satisfied on a stack-heap model, where the heap contains only a single allocated location, $h = \{s(x) \mapsto s(y)\}$. List segment predicate is satisfied on every model where either $s(x) = s(y)$ and the heap is empty, or where the heap is a series of distinct allocated locations $h = \{l_0 \mapsto l_1, ..., l_{n-1} \mapsto l_n\}$ of length at least $n = 1$, where $s(x) = l_0$ and $s(y) = l_n$.

Semantics of boolean connectives are defined as usual. Separating conjunction $\varphi_1 * \varphi_2$ is satisfied on models, for which exist two heaps h_1, h_2, such that

$$(s, h_1) \vDash \varphi_1 \wedge (s, h_2) \vDash \varphi_2 \wedge h_1 \uplus h_2 \neq \perp \wedge h = h_1 \uplus h_2.$$

Septraction is defined similarly,

$$(s, h) \vDash \varphi_1 \mathrel{-\!\!\circledast} \varphi_2 \Leftrightarrow \exists h_1 : (s, h_1) \vDash \varphi_1 \wedge h_1 \uplus h \neq \perp \wedge (s, h_1 \uplus h) \vDash \varphi_2.$$

The operation $\uplus$ is defined as a union of two heaps, but it is defined only for heaps, whose domains are disjoint, and also share only named locations, i.e. locations, which are in the image of s.

3.3 Fragment used for analysis

Although any formula generated by the mentioned grammar is a valid SL formula, I will use just a subset of all valid SL formulae. This fragment has the form $\varphi = \varphi_1 * \varphi_2 * ... * \varphi_n$, where $\varphi_1, ..., \varphi_n$ are atomic formulae for equality, inequality, and for points-to relation. Note that pure atoms are satisfied on empty heap only, which is why separating conjunction is used for both spatial and pure atoms.

4 Shape analysis

Shape analysis is a kind of static analysis that aims to detect the shapes of data structures in the heap, such as linked lists, trees, and variations of those, and to use this knowledge to describe the state of a program's memory in more detail than would be otherwise possible. In [8], the authors propose a method to analyze programs with linked lists using separation logic for better scalability. However, they work with SL of slightly different semantics (pure atoms are satisfied on any heap), and they also implement shape analysis of a minimal language. They also do not use a dedicated solver. My objective is to adapt their method to support at least a part of the C programming language, and to work with SL as implemented by Astral.

4.1 Implementaion

Current state of the plugin that that it can analyze programs without loops, function calls except allocation functions, composite data structures, multiple dereferences on any side of assignments, and global variables. Although the analysis is not very useful at the moment, most of these features can be implemented without any new work with the formulae themselves, merely by preprocessing the CIL AST and adding some heuristics. More on this in Section 5.

Let us start with the type of data that will be stored for each CFG node during the dataflow analysis. Currently, this is a list of SL formulae (`SSL.t list`). Each formula represents a pos-

sible state the program could be in. For example, after a statement `x = malloc(sizeof(void *));`, the program's memory could be described by any of the two formulae $\varphi_1 = (x \mapsto y)$, or $\varphi_2 = (x = \text{nil})$. This expresses the two possible outcomes of calling `malloc` – either the allocation succeeded, and x is now an allocated memory location, or it failed, and `x` is equal to `NULL`. The y in the formula simply represents an arbitrary memory location that x is now pointing to, and it is existentially quantified. This corresponds to the fact that the allocated memory is uninitialized.

The implementation of the analysis itself is done through the `Dataflow2` module API.

- `computeFirstPredecessor` – Currently, this function is implemented as identity.
- `doInstr` – As mentioned, this is the the transfer function for instructions, and therefore much of the logic of the analysis is implemented here. Mind that an instruction in CIL naming convention is any C statement that doesn't affect control flow of the current function. Note that a function call is also considered an instruction. As input, the function gets the instruction itself, and the state of the previous analyzed instruction, and it must return new state for this instruction based on the inputs. For this analysis, I am interested in the following instructions: `LocalInit`, `Set`, `Call`.
  - `LocalInit` is the initialization of a local variable. Initializations of non-pointer variables are currently ignored, and previous state is returned. For pointer variables, the action depends on the initializing value. If the variable is initialized with a call to an allocation function (currently, these are detected simply by name `malloc`, `calloc`, `realloc`), for each previous state, two new states are generated. One of them represents successful allocation, and therefore (<name> $\mapsto$ <fresh>) is appended to it using separating conjunction ($*$). `name` is simply the name of newly initialized variable, and `fresh` is a globally unique (fresh) variable name generated by Astral. I might extend this naming later, when I add support for loops. Any other initialization is currently considered to be an initialization with a constant (`NULL`), and therefore the previous state is extended by (<name> = nil).
  - `Set` is an assignment instruction. Again, this analysis is only interested in assignments to pointer variables, or assignments where the right-hand side contains a dereference. For other assignments, previous state is returned. Simple assignment `a = b;`, where the type of the variables is a pointer type, results in the following change to the previous state. First, all occurences of the variable `a` in the formula are substituted with a new, unique name, and then $a = b$ is appended to the formula with $*$. Assignment of dereferenced variable, `a = *b;`, is more complicated. We must first find the variable, to which `b` is pointing, and then do essentially the same as with simple assignment. To find the variable, to which `b` is pointing to, a transitive closure of equality $C(b)$ is found for `b`, and then the set of all variables that are pointed to by this closure is found simply by iterating through all the atoms of the formula: $T = \{t; \exists c \in C(b) : c \mapsto t\}$. This set T ("target") can be empty – this corresponds to the possibility of dereferencing an invalid pointer in the program. In this case, the analysis is stoppped. T can contain exactly one element $T = \{t\}$, then `a` is substituted with a new variable name as in the first case, and atom $a \mapsto t$ is appended to the substituted formula. This is the computed state. T cannot contain more than a single element, because such formula would be unsatisfiable. Unsatisfiable formulae are filtered out in `doEdge`. For write to dereferenced variable `*a = b`, the single-element set $T = \{t\}$ with the target of `a` is again found, and then the spatial atom $s \mapsto t$ is changed to $b \mapsto t$. `s` is a member of equivalence class $C(a)$. Notice that in this case, no substitutions are done, because no additional equality is added.

- `Call` is a function call instruction. Currently, only allocation functions are supported. The implementation is the same as in allocation during initialization, with the extra step that before creating two states for two outcomes of the allocation, name of the variable `x` in `x = malloc(sizeof(void *));` would be first subtituted in the whole formula with a fresh variable.

- `combinePredecessors` – This function is called when the analysis reaches a CFG node for the second time, and computes new data for this node. `combinePredecessors` gets the old state and the new state as input parameters, and returns these states joined together, as well as information on whether the old state was updated in any way. This is then used by the dataflow algorithm to decide whether to continue updating nodes following this one. Note that the state of a CFG node is internally a list of formulae, but the meaning is a logical disjunction of these (the models of a state for single CFG node is the union of models of all the formulae in the list). In theory, we could simply take the old state φ_{old} and new state φ_{new}, and check the entailment $\varphi_{\text{old}} \models \varphi_{\text{new}}$. If this were true, all models of the new state would have been contained in the old state, and we could have returned that the state had not changed. Otherwise, we would have returned the disjunction of new and old states (internally, a list concatenation). However, this would be imprecise, a single formula in φ_{new}, of which a single model would not be a model of φ_{old} would cause adding the entirety of φ_{new}. Instead, I take the formulae of new state $\varphi_{\text{new}_1}, \varphi_{\text{new}_2}, ..., \varphi_{\text{new}_n}$, and for each one check the entailment $\varphi_{\text{new}_i} \models \varphi_{\text{old}}$. If this is false, φ_{new_i} it is added to φ_{old}, otherwise it is discarded. Finally, if any formulae were added to φ_{old}, the function would return that state for this CFG node was changed.

- `doStmt` – This function decides, whether to continue with analysis upon reaching a statement. This is independent of the actual algorithm for dataflow analysis, which decides when the analysis is complete – that is, when the state of all statements cannot be updated. Currently, this is set to always continue analysis.

- `doGuard` – called when the analysis reaches an `if` statement. The function gets the state from previous node and the condition expression, and returns the states to use in each of the branches. Currently, all conditions that are not in the form `a == b` or `a != b` are considered nondeterministic. When reaching nondeterministic condition, the analysis simply uses the input state as the state for both branches, since it gained no additional information about what is true in the branches. If the condition is `a == b`, then all formulae of input state φ_{in_i} are tested separately using the following method. For "then" branch, each formula is appended with $(a = b)$ using separating conjunction, and checked for satisfiability. Unsatisfiable formulae are not passed into the "then" branch. For the "else" branch, the same method is used, only the formula is appended with $(a \neq b)$. This way, only satisfiable formulae are passed into their respective branches, simplifying further analysis. For the case of condition `a != b`, the algorithm is the same, only the states for the two branches are swapped.

- `doEdge` – called between the updates of nodes, it can modify the state that is sent from previous node to next one. In this function, satisfiability of all formulae of the state is checked, and unsatisfiable formulae are removed from the state. All formulae are also simplified using Astral's `Simplifier.simplify` function, which performs many simple syntactic optimizations of the formula without changing its semantics, such as flattening of nested operations.

The entrypoint of the analysis is the function `run`, which fetches the entrypoint of the input code, sets up the initial state for the analysis (which is just a formula describing the empty

heap, here represented as (nil = nil)). Then it runs the analysis to completion, and prints out the result (a list of formulae for each statement).

4.1.1 Example

The following table contains the analysis results of a simple program. At the end, you can see that the original variable `x` was renamed to a fresh variable. Fresh variables are marked f_n, each having a unique index.

`int *nullptr = NULL;`	$\text{nullptr} = \text{nil}$
`void *x = malloc(sizeof(void *));`	$\text{nullptr} = \text{nil} * x \mapsto f_0$ $\text{nullptr} = \text{nil} * x = \text{nil}$
`if (x == nullptr) {`	
`;`	$\text{nullptr} = \text{nil} * x = \text{nil}$
`} else {`	
`*(void**)x = malloc(sizeof(void *));`	$\text{nullptr} = \text{nil} * x \mapsto f_0 * f_0 \mapsto f_1$ $\text{nullptr} = \text{nil} * x \mapsto f_0 * f_0 = \text{nil}$
`}`	
`; // join of branches`	$\text{nullptr} = \text{nil} * x = \text{nil}$ $\text{nullptr} = \text{nil} * x \mapsto f_0 * f_0 \mapsto f_1$ $\text{nullptr} = \text{nil} * x \mapsto f_0 * f_0 = \text{nil}$
`x = nullptr;`	$\text{nullptr} = \text{nil} * f_2 = \text{nil} * x = \text{nullptr}$ $\text{nullptr} = \text{nil} * f_3 \mapsto f_0 * f_0 \mapsto f_1 * x = \text{nullptr}$ $\text{nullptr} = \text{nil} * f_4 \mapsto f_0 * f_0 = \text{nil} * x = \text{nullptr}$

5 Future work

Let us address each of the deficiencies of the current implementation, and possible ways to solve them. First, there is the issue of loops. In the current state of the plugin, there is actually no problem with analysis of loops in general, only with pointer manipulation inside loops. For these programs, the analysis would not terminate.

The first issue is with initializations within a loop. Currently, variable names inside formulae used for analysis are taken from the source code without any renaming, therefore when the analysis reaches an initialization twice, it produces an invalid state, because it assumes that such a variable does not exist, but it in fact does, from the previous iteration of the loop. This can be solved in two ways. One option is to always do a substitution of the name of variable being initialized, and the other is to preprocess the code in such a way, that the initialization will always be outside of the loop. This will require renaming variables in the CIL AST representation, since pulling a variable to outer scope might cause a name collision. Ideally, all variable initializations should be pulled into the top scope of the function, to avoid further problems with conflicting variable names in nested scopes, which I have not yet encountered.

Not considering initializations, other pointer manipulations done inside a loop will technically produce correct formulae, but these will grow without any limits as the analysis progresses. For example, consider a program that constructs a linked list using a while loop with nondeterministic condition. In each iteration, the formula would get longer by a single points-to atom. This

can be solved using the list segment atom $\text{ls}(x, y)$. Since it represents a sequence of any size, we can abstract the growing chain of pointers into a single list segment. Note that this has limitations, when abstracting the formula $x \mapsto y * y \mapsto x * ...$ into $\text{ls}(x, z) * x \neq z * ...$, y cannot be referenced in the rest of the formula, because we would lose the information that it points to z. Also, list segment represents an *acyclic* linked list, which means that $x \neq y \neq z$ must hold. Introducing list segments into the state formulae will also mean that most other parts of the analysis will have to be extended to accommodate the possibility of x pointing to an unnamed location inside the list segment. With this, the analysis should be able to process basic programs working with linked lists.

Another shortcoming of the current implementation is the limited shape of statements supported by the analysis. However, by supporting the four basic statements, all statements that are a combination of these can be broken down into a sequence of the basic ones:

```
a = b;
a = *b;
*a = b;
a = malloc(...);
```

Take, for example, the statement `*a = **b;`. We can break it down into a sequence of the following statements, using fresh variables:

```
tmp_1 = *b;
tmp_2 = *tmp_1;
*a = tmp_2;
```

This can be done algorithmically, in another preprocessing step on the CIL AST.

Another problem is that currently we can only work with simple pointer types, but for analysis of real programs, the analysis must support structs implementing the linked list ADT. I propose that in yet another preprocessing pass, accesses to the "next" field of the linked list struct are to be rewritten to a simple dereference, whereas accesses to other fields of the struct would be simply ignored. Of course, accesses to other fields over a pointer (eg. `list->data = 5;`) would be checked, whether `list` is allocated, without actually changing the formula. Such a rewrite might look like this:

```
list->next = malloc(...);
tmp = list->next;
tmp2 = list2->data;
```

could be transformed to

```
*list = malloc(...);
tmp = *list;
// this line would be omitted, only list would be checked, whether it is allocated
```

The question of how to detect, which field of a struct is the "next" field probably does not have a perfect solution, but a heuristic is easy to implement. If a struct has a single field with a type of pointer to itself, this field would be considered the "next" field. If more than one self-referential pointer is found, it would indicate another data structure, such as a binary tree, or doubly linked list. Another option is to rely on user to annotate the structs and their fields manually.

Another missing feature is the analysis of function calls, by which I mean other than allocation functions. Analysis of the `free` function is also not implemented, but it should only require calling existing code. It should suffice to find the target of the freed variable by looking at

points-to atoms from the transitive closure of freed variable, and then to simply remove the points-to atom. User-defined functions are, however, a completely different issue. The original paper [8] does not include functions in its minimal language, so the following method is just my proposition. When calling a function, variables inside the state formula passed as arguments will be renamed to their corresponding parameter names, and this state will be passed as the initial state for the analysis of the function. After returning, the returned value would be renamed back to the variable name the call result was assigned to. This would require all variable names to be unique across functions, and when leaving a function, all of its local variables would have to be renamed to fresh names – this corresponds to a memory leak when returning from a function, where a local variable holds a pointer to allocated memory. Even simpler way to deal with function calls could be to inline everything into `main`. Note that this would not be applicable to recursive functions.

Global variables should not be hard to implement, once analysis of functions is already implemented. Global variables would simply have the same name across all functions they are used in. Their definitions could then simply be moved to the beginning of the entrypoint function (`main`). When calling another function, global variables used in the state formula would be passed into the called function without any renaming. Static variables declared inside functions could be first converted to global variables.

6 Conclusion

Even though the current implementation of the analysis is not capable of processing real-world programs, it shows promise that it will be extendable to cover most C language features to some extent. In theory, most of the work on the analysis itself should already be done, the main part that remains to be implemented is the abstraction to list segments. Besides that, most other work lies in preprocessing the input code to simpler form so that can be analyzed by the existing implementation.

Bibliography

[1] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", *SIGPLAN Not.*, vol. 42, no. 6, p. 89, Jun. 2007, doi: 10.1145/1273442.1250746.

[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: a fast address sanity checker", in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, in USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[3] 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.

[4] 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.

[5] "Frama-C API documentation". [Online]. Available: https://frama-c.com/download/frama-c-27.1-Cobalt-api.tar.gz

[6] T. Dacík, "A Decision Procedure for Strong-Separation Logic", 2022. [Online]. Available: https://www.fit.vut.cz/study/thesis/25151/

[7] T. Dacík, "Astral solver". [Online]. Available: https://github.com/TDacik/Astral

[8] 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.