Modular static analysis of string manipulations in C programs

Matthieu Journault, Antoine Miné, Abdelraouf Ouadjaout

Modular static analysis of string manipulations in C programs*

Matthieu Journault, Antoine Miné, Abdelraouf Ouadjaout

Sorbonne Université, CNRS,
Laboratoire d'Informatique de Paris 6, LIP6,
F-75005 Paris, France
(matthieu.journault||antoine.mine||abdelraouf.ouadjaout)@lip6.fr

Abstract. We present a modular analysis able to tackle out-of-bounds accesses in C strings. This analyzer is modular in the sense that it infers and tabulates (for reuse) input/output relations, automatically partitioned according to the shape of the input state. We show how the inter-procedural iterator discovers and generalizes contracts in order to improve their reusability for further analysis. This analyzer was implemented and was able to successfully analyze and infer relational contracts for functions such as `strcpy`, `strcat`.

1 Introduction

Abstract interpretation [9] enables the development of sound static analyzers that infer and prove invariants on the set of states reachable in a program. Consider for instance the `strcpy` function in C, shown in Program 1.1. This function is often called and may cause out-of-bounds errors. Therefore the implementation of a modular static analyzer able to infer and prove contracts on such functions without losing precision would yield a scalable analyzer able to prove the absence of buffer overruns in C projects manipulating strings.

```
while (*q != '\0') {
  *p = *q;
  p++;
  q++;
}
*p = *q;
```

Program 1.1: `strcpy`

In a C string, a `'\0'` character designates the end of the string. Henceforth the length of a string is defined to be the index of the first `'\0'` character appearing in the string. As emphasized in Program 1.1, the correctness of a string manipulating program (in the sense that it does not yield an out of memory access) depends upon the length and the allocated size of the buffer in which it is contained. Therefore in the fashion of [24] we summarize strings by two values: the position of the first `'\0'` character and the buffer size.

The fragment of C on which we want to perform modular analysis supports string manipulations, unions, structures, arrays, memory allocations (static and dynamic), pointer casts, function calls, ... Accordingly we need to build our

* This work is supported by the European Research Council under Consolidator Grant Agreement 681393 – MOPSA.

analyzer, that manipulates predicates and can perform modular analysis, upon an existing analyzer able to deal with low level features of `C`.

An analyzer computing invariants by induction on the syntax of programs requires abstract transformers for function calls. A straightforward way to achieve this, provided that there is no recursivity, is to analyze the body of the function at each call site. Therefore a way to improve scalability is to design modular analyzers able to reuse previous analysis results so that reanalysis is not always needed (as emphasized in [11]). As an example, in a project containing an incrementation function, we want to be able to express and to infer that $\forall x, \texttt{incr}(x) = x + 1$. Once this relation discovered, no further analysis of the body of `incr` is required. Abstract interpretation is always sound and inferred invariants describe an over-approximation of the reachable set of states. Therefore the use of input/output relations discovered on statements must yield an over-approximation. Nonetheless we do not want to give up too much precision to achieve scalability. This was done by using classical techniques to express input/output relations on numerical variables as performed in [11], partitioning these relations according to symbolic conditions in the abstract state as proposed by Bourdoncle [4], and generalizing them using widening operations.

By mixing the idea of representing a string as its length using a numeric abstraction and input/output relations, our analyzer is able to handle the `strcpy` example. More precisely, consider that `char* p` points to some `char[10] dest` string and `char* q` points to some `char[20] src` string with `dest` $\neq$ `src`, furthermore let variables $o_{\texttt{q}}$, $o_{\texttt{p}}$ denote the initial offset of `p` and `q`, variable $l_{\texttt{src}}$ codes for the length of `src`, and variable $a_{\texttt{dest}}$ denotes the size of the allocated memory of the string pointed to by `dest`. Our analyzer is able to prove that if $l_{\texttt{src}} < a_{\texttt{src}}$ and $l_{\texttt{src}} - o_{\texttt{q}} < a_{\texttt{dest}} - o_{\texttt{p}}$ then no out of bounds access are performed. Moreover enabling modular analysis would yield that $l_{\texttt{src}} = l'_{\texttt{src}}$ and $l_{\texttt{dest}} = l'_{\texttt{src}}$ where primed variables (resp. unprimed variables) denote the state at the beginning (resp. at the end) of the analysis of the body of the function. Therefore these two relations state that the length of `src` was not modified by the call to `strcpy`, while the length of `dest` is now that of `src`.

Outline. Section 2 describes the subset of `C` we wish to analyze, Sect. 3 defines a low-level `C` abstraction upon which our analyzer is based, Sect. 4 details the String abstract domain, Sect. 5 outlines the lifting of our analyzer to a modular analysis, Sect. 6 contains a few remarks on the implementation of the analyzer. Finally Sect. 7 gives an overview of related works, while Sect. 8 concludes.

Contributions. The main contributions of this article are : (1) The development of a static analyzer able to reason on low level `C`, while performing higher level abstractions (such as the String domain that will be presented thereafter) (2) The lifting of this analyzer to a precise modular framework based on numerical input/output relations [11], partitioning [4] and input generalizations.

$$
\begin{array}{lll}
\textit{int-type} & \triangleq & \textbf{s8} \mid \textbf{s16} \mid \textbf{s32} \mid \textbf{s64} \\
& & \mid \textbf{u8} \mid \textbf{u16} \mid \textbf{u32} \mid \textbf{u64} \\
\textit{scalar-type} & \triangleq & \textit{int-type} \mid \textbf{ptr} \\
\textit{type} & \triangleq & \textit{scalar-type} \\
& & \mid \textit{type}[n] \quad n \in \mathbb{N} \\
& & \mid \textbf{struct}\{u_0 : \textit{type}, \ldots, u_{n-1} : \textit{type}\} \\
& & \mid \textbf{union}\{u_0 : \textit{type}, \ldots, u_{n-1} : \textit{type}\}
\end{array}
$$

$$
\begin{array}{lll}
\textit{lval} & \triangleq & *_{\textit{scalar-type}}\, \textit{expr} \mid v \in \mathcal{V} \\
\textit{expr} & \triangleq & \textit{cst} \quad \textit{cst} \in \mathbb{N} \\
& & \mid \texttt{\&}\textit{lval} \\
& & \mid \textit{expr} \diamond \textit{expr} \quad \diamond \in \{+, \leq, \ldots\} \\
\textit{stmt} & \triangleq & v = \texttt{malloc}(e) \\
& & \quad v \in \mathcal{V}, e \in \textit{expr} \\
& & \mid \textit{type}\ v \quad v \in \mathcal{V} \\
& & \mid \ldots
\end{array}
$$

Fig. 1: The syntax of the `C--` subset of `C`.

2 Syntax and concrete semantics

Syntax. We will thereafter call `C--` the language defined in Figure 1 and denote by $\mathcal{V}$ a set of variables. The description of Figure 1 omits some classical statements but make precise some low-level features of the language. Note moreover that *int-types* are denoted by their signededness (**s** for signed integers, **u** for unsigned integers) and their length in bits, instead of `char` or `unsigned long`. This transformation is made before the analysis, and depends on the platform. Moreover, in order to simplify the presentation we will consider strings as arrays of **u8** (or **unsigned char**), results can be easily extended to arrays of **s8** (**char**).

Cells. Our `C`-like language features a rich type system. In a classic way, we will present the semantics of operations on scalar data-types: integers of various size and pointers, and reduce structured data-types, such as arrays, struct and unions, malloced blocks, to collections of scalar objects, we call *cells*. A simple solution would be to use the type of a structured variable and decompose it statically into such collections; left-values thus become access paths. Unfortunately, this static view does not hold for programs that abuse the type system and access some block of memory with various types, which is possible (and even common) in `C` using union types and pointer casts. One solution would be to model the memory as arrays of bytes or even bits, and synthesize non-byte access (for instance, reading a 16-bit integer `a` would be expressed as `a[0]+256*a[1]`), but such a complex modeling would put a great strain on numeric abstract domains and cause huge precision losses. We thus rely on previous work [18], that proposes to model memory blocks as collections of (possibly multi-byte) scalar cells, that are inferred and maintained dynamically during the analysis, according to the memory access pattern effectively employed by the program at run-time. For our purpose, we can assume that all memory accesses have the form $*_\tau e$, where τ is a scalar type and e is a pointer expression using pointer arithmetic at the byte level (this reduction can be performed statically as a pre-processing).

Remark 1. In addition to the definitions of Figure 1, we assume that we are given a function $\textit{typeof} \in (\mathcal{V} \to \textit{type})$. The type of a variable is given by its declaration in a `C--` program. Moreover we assume given a *sizeof* function from *type* to $\mathbb{N}$ that gives the size in bytes of each type (e.g. $\textit{sizeof}(\textbf{int}) = 4$).

A cell denotes an addressable group of bytes to store a scalar value, it is represented by a base variable (V), an integer coding for the offset of the cell (o), and the type of the cell (t). Therefore we define the following set of cells: $\mathcal{C}ell \triangleq \{\langle V, o, t\rangle \mid V \in \mathcal{V},\ t \in \mathit{scalar\text{-}type},\ 0 \leq o \leq \mathit{sizeof}(\mathit{typeof}(V)) - \mathit{sizeof}(t)\}$. By construction $\mathcal{C}ell$ represents the set of all addressable memory locations. The abstract states we will build contain a subset of those cells. Cells might denote overlapping portions of the memory. In such cases the underlying state satisfies every constraint implied by a cell : cells are understood conjunctively. Therefore removing cells induces a loss of information. A key aspect of [18] we reuse is that new cells from $\mathcal{C}ell$ are added to the current environment dynamically to account for the access patterns encountered during the analysis, in a flow-sensitive way. As we do not rely on static type information, which can be misleading in C, we can handle union types, type-punning, and untyped allocated blocks transparently.

Concrete semantic. We will not detail here the complete concrete semantic of the `C--` language, however we give a definition of the set of concrete environments using cells, noted $\mathcal{E}$. An environment is a set of cells C and a function ρ mapping each cell to a value. A value can be either a numerical value or a pointer. A pointer is represented by: the base variable towards which it points and its offset. The set of pointer $\mathcal{P}tr$ is augmented with two special values: the **NULL** pointer and the **invalid** pointer : $\mathcal{P}tr \triangleq (\mathcal{V} \times \mathbb{Z}) \cup \{\mathbf{NULL}, \mathbf{invalid}\}$

$$\mathcal{E} \triangleq \bigcup_{C \subseteq \mathcal{C}ell} \{\langle C, \rho\rangle \mid \rho \in R \triangleq C \to (\mathbb{N} \cup \mathcal{P}tr)\}$$

3 Cell abstract domain

Let us consider the Cell abstraction [18], an abstract domain able to abstract the semantic of `C` programs manipulating pointers. This abstract domain comes with an abstract interpreter that can successfully analyze `C` programs with no recursion and no dynamic memory allocation. The abstraction we propose here is built upon the cell abstract domain, it extends this domain so as to handle dynamic allocations and higher level string manipulations.

Pointers bases. When $C \subseteq \mathcal{C}ell$ is a set of cells, we define $\overline{C}$ to be the set of cells denoting pointers : $\overline{C} \triangleq \{\langle V, o, t\rangle \in C \mid t = \mathbf{ptr}\}$. Upon this we define $\mathcal{P}_C = \overline{C} \to \wp(\mathcal{V} \cup \{\mathbf{NULL}, \mathbf{invalid}\})$. $\mathcal{P}_C$ represents the possible memory locations pointed to by cells representing pointers (note that $\mathcal{P}_C$ only accounts for the base variable that is pointed to and not for the offset).

Numerical domain. We assume that for any set of variables $\mathcal{V}$ we are given a numerical domain $N^\sharp_\mathcal{V}$, abstracting $\wp(\mathcal{V} \to \mathbb{N})$ with concretization function $\gamma_\mathcal{V} \in N^\sharp_\mathcal{V} \to \wp(\mathcal{V} \to \mathbb{N})$. For example we can use the polyhedra domain [15] or the interval domain [10]. These domains come with an environment change operator $\square_{|\mathcal{V}}$ such that: $\forall \mathcal{V}'$, if $S^\sharp \in N^\sharp_{\mathcal{V}'}$ then $S^\sharp_{|\mathcal{V}} \in N^\sharp_\mathcal{V}$. $S^\sharp_{|\mathcal{V}}$ is obtained by removing all

variables not in $\mathcal{V}$ and adding all variables in $\mathcal{V}$ but not in $\mathcal{V}'$ (with unconstrained value), so that the result is defined exactly over the variable set $\mathcal{V}$. Furthermore we assume given a function $\mathbf{range}(x, S^\sharp_\mathcal{V})$, yielding an interval of $\mathbb{N}$ containing all concrete values associated to variable x in $N^\sharp_\mathcal{V}$. For any subset $C \subseteq \mathcal{C}ell$ we can therefore rely on a numerical abstraction $N^\sharp_C$ abstracting $\wp(C \to \mathbb{N})$. We give the numerical domain of our abstraction a double role:

- For a cell containing a pointer, the variable (from the numerical domain) assigned to this cell codes for possible offsets of the pointer (thus paired with information from $\mathcal{P}_C$ we will describe completely the pointer contained in the cell)
- For other cells (containing e.g. a **u8**, a **s32**) the variable (from the numerical domain) codes for values contained in the cell.

Abstract states. We define the domain $\mathcal{D}^\sharp_m$ with concretization $\gamma_m \in \mathcal{D}^\sharp_m \to \mathcal{E}$ as:

$$\mathcal{D}^\sharp_m \triangleq \{\langle C, R^\sharp, P\rangle \mid C \subseteq \mathcal{C}ell,\ R^\sharp \in N^\sharp_C,\ P \in \mathcal{P}_C\}$$

$$\gamma_m\langle C, R^\sharp, P\rangle \triangleq \langle C, \{\rho' \in R, \exists \rho \in \gamma_C(R^\sharp),\ \forall c = \langle V, o, t\rangle \in C,$$

$$\begin{cases} \rho'(c) = \rho(c) & \text{if } t \neq \textbf{ptr} \\ \rho'(c) = \langle p, \rho(c)\rangle & \text{if } t = \textbf{ptr} \wedge p \in P(c) \cap \mathcal{V} \\ \rho'(c) = p & \text{if } t = \textbf{ptr} \wedge p \in P(c) \setminus \mathcal{V} \end{cases} \}\rangle$$

Example 1. Consider the abstract state: $S^\sharp = \langle\{\langle \mathtt{a}, 0, \textbf{u64}\rangle\}, \{\langle \mathtt{a}, 0, \textbf{u64}\rangle = 2^{32} + 2\}, \emptyset\rangle$. Moreover we assume that due to some cast operations, cells $\{\langle \mathtt{a}, 0, \textbf{u32}\rangle\}$ and $\{\langle \mathtt{a}, 4, \textbf{u32}\rangle\}$ are needed (imagine for example the encoding in little-endian of a pair of **u32** as a **u64**). $S^\sharp$ is equivalent to: $\langle\{\langle \mathtt{a}, 0, \textbf{u64}\rangle, \langle \mathtt{a}, 0, \textbf{u32}\rangle, \langle \mathtt{a}, 4, \textbf{u32}\rangle\}, \{\langle \mathtt{a}, 0, \textbf{u64}\rangle = 2^{32} + 2, \langle \mathtt{a}, 0, \textbf{u32}\rangle = 2\ (= (2^{32} + 2) \mod 2^{32})), \langle \mathtt{a}, 4, \textbf{u32}\rangle = 1\ (= (2^{32} + 2)\ /2^{32})\}, \emptyset\rangle$.

```
int a = 1;      1
int p = &a;     2
*p = *p + 1;    3
```

Program 1.2: Dereferencing

Abstract operators and abstract transformers. Abstract operators (join, meet, widening) are defined by first unifying the operands, and then performing the operation in the underlying unified numerical domain and pointer map. The unification operator transforms two abstract elements into abstract elements with the same set of cells. This is done by adding cells in both elements so that the resulting set of cells in both elements is the union of the initial sets of cells. We do not give here the definition of all the abstract transformers operating on our abstract states, however the following example emphasizes how an abstract state is modified by expressions and statements of the `C` language. In particular, we note that when cells are available, most expressions are treated as expressions on a language where cells are the variables. When cells mentioned in the expressions are not available, they are added to the set of cells of the abstract state, by collecting information available in the overlapping cells, such as joining two byte-cells to synthesize the initial value of a new **u16**-cell at the same position.

Example 2. Consider Program 1.2, starting from $\top = \langle \emptyset, \emptyset, \emptyset \rangle$. The first statement requires the existence of the cell $\mathbf{a} = \langle \mathtt{a}, 0, \mathbf{s32} \rangle$. The set of cells constrained by our abstract state is dynamically updated to mention $\mathbf{a}$, yielding: $\langle \{\mathbf{a}\}, \emptyset, \emptyset \rangle$, then we rewrite the statement in the following manner: `a = 1`. We execute this statement in the underlying numerical domain, and get: $\langle \{\mathbf{a}\}, \{\mathbf{a} = 1\}, \emptyset \rangle$. The second statement adds a new cell $\mathbf{p} = \langle \mathtt{p}, 0, \mathbf{ptr} \rangle$ and an element to the pointer map: $\langle \{\mathbf{a}, \mathbf{p}\}, \{\mathbf{a} = 1, \mathbf{p} = 0\}, \{\mathbf{p} \mapsto \{\mathtt{a}\}\} \rangle$. Note that $\mathbf{p} = 0$ codes for the value of the offset of pointer `p`. Finally the expression `*p` of the third statement is evaluated by following the P component of the abstract state, therefore the statement is transformed into `a = a + 1`. Thus yielding: $\langle \{\mathbf{a}, \mathbf{p}\}, \{\mathbf{a} = 2, \mathbf{p} = 0\}, \{\mathbf{p} \mapsto \{\mathtt{a}\}\} \rangle$. Henceforth, in order to clarify the presentation, $\mathbf{a}$ denotes $\langle a, 0, \tau \rangle$ when τ is the declared type of variable `a`.

4 String abstract domain

4.1 Domain definition

The introductory example shows that describing a string by a set of cells (one cell per character of the string) was usually not necessary to prove the absence of buffer overrun in string manipulations. Therefore we propose to add to our existing low-level abstraction of `C--`, an abstraction of strings that sums up all of its characters into two variables, one coding for the length of the string and the other for the allocated size of the buffer in which it is contained. Memory blocks will therefore be abstracted either by the cell abstract domain or by the string abstract domain. In order to simplify the presentation we assume given a set of memory locations $\mathfrak{V}$ for which we will use a string summary, however this set can be dynamically modified and reductions could be proposed in order to store information on some memory locations in both the String domain and the Cell domain. We assume that for each memory location $s \in \mathfrak{V}$, we are given two variables denoted s_l and s_a. Those variables code for the length and the allocated size of the string, they will be added to the numerical domain of the cell abstract domain so that we are able to describe relations between length of variables and offsets of pointers. In the following $\mathfrak{V}^\star$ denotes $\bigcup_{s \in \mathfrak{V}} \{s_a, s_l\}$, this is the set of all numerical variables needed to describe strings in $\mathfrak{V}$.

Definition of the String abstract domain. We define the String abstract domain to be: $\mathcal{S}^\sharp_m \triangleq \{\langle C, R^\sharp, P \rangle \mid C \subseteq \mathcal{C}ell \setminus \{\langle V, \_, \_ \rangle \mid V \in \mathfrak{V}\},\ R^\sharp \in N^\sharp_{C \cup \mathfrak{V}^\star},\ P \in \mathcal{P}_C\}$. This abstract domain is ordered by the same relation as the cell abstract domain: $\sqsubseteq_{S^\sharp_m} \triangleq \sqsubseteq_{D^\sharp_m}$. We recall that $\sqsubseteq_{\mathcal{D}_m}$ will test the inclusion of the two numerical domains once cell sets have been unified, therefore our definition of $S^\sharp \sqsubseteq_{S^\sharp_m} S^{\sharp\prime}$ amounts to verifying that the constraints on the string variables (s_l and s_a) are stronger in the left member of the inequality.

Galois connection with the Cell abstract domain. The String abstract domain is an abstraction of the Cell abstract domain. Indeed we forget information that do

not help us track the position of the first `'\0'` character. We define the Galois connection between the Cell domain and the String domain using two functions : **to_cell** and **from_cell**. The **to_cell**$(s, S^\sharp)$ function computes the range of s_l in the numeric abstract domain, for each possible length value we set the cells placed before (resp. at) the length to $[1; 255]$ (resp. 0), this yields an abstract element per possible value in the range, those are then joined. Conversely **from_cell**$(s, S^\sharp)$ computes the minimum length value (the index of the first cell whose range contains 0), and the maximum length value (the index of the first cell whose range is exactly $\{0\}$), finally those constraints are added to the numerical domain. If a string does not contain any `'\0'` character, we define its length to be the allocated size of the buffer it is contained in. Both functions can be found in Appendix A.1. With $\mathfrak{V} = \{s_0, \ldots, s_{n-1}\}$, we can define:

$$\gamma_{S_m^\sharp, D_m^\sharp}(S^\sharp) = \textbf{to\_cell}(s_0, \ldots, (\textbf{to\_cell}(s_{n-1}, S^\sharp)) \ldots)$$
$$\alpha_{S_m^\sharp, D_m^\sharp}(S^\sharp) = \textbf{from\_cell}(s_0, \ldots, (\textbf{from\_cell}(s_{n-1}, S^\sharp)) \ldots)$$

Example 3. Consider the string abstract elements $\langle \emptyset, \{s_l = 2, s_a = 4\}, \emptyset \rangle$ when *sizeof*(*type*(s)) $= 4$. We have: $\gamma_{S_m^\sharp, D_m^\sharp} = \langle \{\langle s, 0, \textbf{u8}\rangle, \langle s, 1, \textbf{u8}\rangle, \langle s, 2, \textbf{u8}\rangle\}, \{\langle s, 2, \textbf{u8}\rangle = 0, \langle s, 0, \textbf{u8}\rangle \neq 0, \langle s, 1, \textbf{u8}\rangle \neq 0\}, \emptyset\} \rangle$. The corresponding concrete state is the set of states in which there is a memory location where the first two bytes are non zero bytes, the third byte is set to zero and the fourth byte is unconstrained.

Remark 2. The interest of the definition of $\gamma_{S_m^\sharp, D_m^\sharp}$ and $\alpha_{S_m^\sharp, D_m^\sharp}$ is twofold: it enables us to define the semantic of the String abstract domain, but we also note that both functions **to_cell** and **from_cell** are computable. Therefore the set of memory locations dealt with by each domain can easily evolve during the analysis. Moreover with $\gamma_{S_m^\sharp, D_m^\sharp}$ and $\alpha_{S_m^\sharp, D_m^\sharp}$ being both computable, we can define a reduction operator between the String abstract domain and the Cell abstract domain. For efficiency reasons we can remove some information from the Cell domain, knowing that information from the String domain can be brought back to the Cell domain. This situation is similar to a reduction between octagons and the, strictly less expressive, interval domain as proposed in [12].

4.2 Operators and transformers

Operators. As for the definition of the $\sqsubseteq_{S_m^\sharp}$ operator, the join ($\sqcup_{S_m^\sharp}$), meet ($\sqcap_{S_m^\sharp}$) and widening ($\triangledown_{S_m^\sharp}$) of two abstract elements is defined by applying the according operator in the underlying numerical abstract domain (after the addition on both sides of potentially missing variables and the unification of the set of cells).

Example 4. Consider Program 1.1 of the introductory example where *typeof*(`p`) = *typeof*(`q`) = **u8***, if $S_1^\sharp = \langle \{\mathbf{p}, \mathbf{q}\}, \{\mathbf{p} = 0, \mathbf{q} = 0, \mathtt{src}_l \geq 0, \mathtt{src}_a \geq \mathtt{src}_l, \mathtt{dest}_l \geq 0, \mathtt{dest}_a \geq \mathtt{dest}_l\}, \{\mathbf{p} \mapsto \{\mathtt{dest}\}, \mathbf{q} \mapsto \{\mathtt{src}\}\} \rangle$ is the abstract state from which we start the analysis then $S_2^\sharp = \langle \{\mathbf{p}, \mathbf{q}\}, \{\mathbf{p} = 1, \mathbf{q} = 1, \mathtt{src}_l \geq 1, \mathtt{src}_a \geq \mathtt{dest}_l, \mathtt{dest}_l \geq 1, \mathtt{dest}_a \geq 1, \mathtt{dest}_a \geq \mathtt{dest}_l\}, \{\mathbf{p} \mapsto \{\mathtt{dest}\}, \mathbf{q} \mapsto \{\mathtt{src}\}\} \rangle$ is the abstract state after one analysis of the body of the `while` loop (constraint

$$\textbf{before}(e, s, S^\sharp) = \{([1;255], \langle C, R^\sharp \sqcap \{0 \leq e, e < s_l, e < s_a\}, P\rangle\}$$

function	tests on offset	evaluation
before	$0 \leq o \wedge o < l \wedge o < a$	$[1;255]$
at	$0 \leq o \wedge o = l \wedge o < a$	0
after	$0 \leq o \wedge o > l \wedge o < a$	$[0;255]$
eerror	$o > a \vee o < 0$	$\emptyset$

Fig. 2: Evaluation of a dereferencing.

$\texttt{src}_a \geq \texttt{dest}_l$ comes from the fact that we collect error free executions). Therefore our analyzer has to perform the join of those two abstract states before reanalyzing the body of the loop. $S_1^\sharp \sqcup_{S_m^\sharp} S_2^\sharp = \langle\{\mathbf{p}, \mathbf{q}\}, \{\mathbf{p} = \mathbf{q}, \mathbf{p} \leq 1, \mathbf{p} \geq 0, \mathbf{p} \leq \texttt{dest}_l, \texttt{dest}_a \geq \texttt{dest}_l, \mathbf{p} \leq \texttt{src}_l, \mathbf{p} \leq \texttt{src}_a\}, \{\mathbf{p} \mapsto \{\texttt{dest}\}, \mathbf{q} \mapsto \{\texttt{src}\}\}\rangle$.

The state transformations induced on our abstract state by the semantic of the `C` language is mainly dealt with by the Cell abstraction. In order to ease the presentation of the relation between the Cell abstraction and the String abstraction, we add expressions of the form `@[`v`,`e`]` with $e \in$ *expr* and $v \in \mathcal{V}$ to the `C--` language. Such expressions denote pointers to variable v, with offset e: `((char *) &`v`) +` e.

Example 5. We want to perform the analysis of the statement

$$\texttt{stmt} = \texttt{*}_{\mathbf{u8}}\texttt{t = *}_{\mathbf{u8}}\texttt{(p + *}_{\mathbf{s32}}\texttt{(\&u + 2))}$$

(where *typeof*(p) = *typeof*(t) = **u8***) in the following abstract state: $S^\sharp = \langle\{\mathbf{p}, \langle \mathtt{u}, 2, \mathbf{s32}\rangle, \mathbf{t}\}, R^\sharp, \{\mathbf{p} \mapsto s', \mathbf{t} \mapsto s\}\rangle$ where $R^\sharp$ is a numerical abstract state built from the set of constraints we do not need to explicit for this example. The Cell abstraction rewrites `stmt` into: $\texttt{*}_{\mathbf{u8}}\texttt{@[}s\texttt{,t] = *}_{\mathbf{u8}}\texttt{(@[}s'\texttt{,p +} \langle \mathtt{u}, 2, \mathbf{s32}\rangle\texttt{])}$. The operations that remain to be defined are therefore:

$\mathbb{S}^\sharp[\![\texttt{*}_\tau\texttt{@[}s\texttt{,}e_1\texttt{] = }e_2]\!](S^\sharp)$ where $s \in \mathfrak{V}, e_1 \in$ *expr*$, e_2 \in$ *expr*$, \tau \in$ *scalar-type*
$\mathbb{E}^\sharp[\![\texttt{*}_\tau\texttt{@[}s\texttt{,}e\texttt{]}]\!](S^\sharp)$ where $s \in \mathfrak{V}, e \in$ *expr*$, \tau \in$ *scalar-type*

Abstract evaluation. Let us first consider the evaluation of the dereferencing of a pointer to a string. The analyzer we want to define performs partitioning on the abstract state during the evaluation of expressions, therefore evaluation results are pairs (evaluated expression × abstract state). This set is understood disjunctively and greatly improves the precision of the analyzer. The result of an evaluation is therefore a finite element of $\wp(exp \times S_m^\sharp)$. Five cases can be distinguished during the evaluation of $\texttt{*}_\tau\texttt{@[}s\texttt{,}e\texttt{]}$:

- **before**: $\tau =$ **u8** and `@[`s`,`e`]` points before the first `'\0'` character. In this case the evaluation can yield any character that is not `'\0'`.
- **at**: $\tau =$ **u8** and `@[`s`,`e`]` points at the first `'\0'` character. In this case the evaluation yields `'\0'`.
- **after**: $\tau =$ **u8** and `@[`s`,`e`]` points after the first `'\0'` character. In this case the evaluation can yield any character.
- **eerror**: $\tau =$ **u8** and `@[`s`,`e`]` points after the end of the allocated memory. In such a case we generate an **out_of_bounds** error.

function	tests on offsets	tests on rhs	transformation
set0	$o \geq 0 \wedge o \leq l \wedge o < a$	$c = 0$	$l \leftarrow o$
setnon0	$o \geq 0 \wedge o = l \wedge o < a$	$c \neq 0$	$l \leftarrow [o+1; a]$
unchanged	$o \geq 0 \wedge o < l \wedge o < a$	$c \neq 0$	
unchanged	$o \geq 0 \wedge o > l \wedge o < a$	$\top$	
l_unchanged	$o \geq 0 \wedge o > l \wedge o + r \leq a$	$\top$	
forget	$o \geq 0 \wedge o \leq l \wedge o + r \leq a$	$\top$	$l \leftarrow [o; a]$
serror	$o + r > a \vee o < 0$	$\top$	**out_of_bounds**

Fig. 3: Summary of transformations.

– $\tau \neq$ **u8**, in this case we over-approximate the evaluation by the range of the type τ.

Figure 2 summarizes those cases and gives the example of the **before** function. In this table o is the offset of the pointer, l and a are the length and the allocated size of the string. The definition of these functions can be found in in Appendix A.2. We can now define $\mathbb{E}^\sharp[\![*_{\tau=\textbf{u8}}\texttt{@[}s\texttt{,}e\texttt{]}]\!](S^\sharp) = \bigcup\{\textbf{before}(e', s, S^\sharp) \cup \textbf{at}(e', s, S^\sharp) \cup \textbf{after}(e', s, S^\sharp) \cup \textbf{eerror}(e', s, S^\sharp) \mid (e', S^\sharp) \in \mathbb{E}^\sharp[\![e]\!](S^\sharp)\}$ and $\mathbb{E}^\sharp[\![*_{\tau\neq\textbf{u8}}\texttt{@[}s\texttt{,}e\texttt{]}]\!](S^\sharp) = \{(\mathit{range}(\tau), S^\sharp)\}$.

Abstract transformations. In order to complete the definition of our abstract interpreter, we need to provide the abstract semantic of an assignment in a string $*_\tau\texttt{@[}s\texttt{,}e_1\texttt{]} = e_2$. We can distinguish 6 cases in such an assignment:

– **set0**: $\tau =$ **u8** and a character that appears before the first `'\0'` is assigned to `'\0'`, in which case we need to set the variable coding for the length of the string to its new value (the offset of the pointer to the string).
– **setnon0**: $\tau =$ **u8** and the first `'\0'` is replaced with a non-`'\0'` character, in which case we need to set the variable coding for the length of the string to its new value (it can be anything greater than the offset of the pointer to the string).
– **unchanged**: $\tau =$ **u8** and we are performing an assignment that does not change the position of the first `'\0'` character. Either because we are replacing a character placed before the first `'\0'` character by a non-`'\0'` character, or because we are assigning a character after the position of the first `'\0'` character.
– **l_unchanged**: $\tau \neq$ **u8** and we are performing an assignment that does not change the position of the first `'\0'` character: the only modified characters are placed after the first `'\0'` character.
– **forget**: $\tau \neq$ **u8** and the offset of the pointer is less than the length of the string, in this case the position of the first `'\0'` character is greater than the offset of the pointer.
– **serror**: The writing generates an out of bounds, in which cases we generate an **out_of_bounds** warning.

Figure 3 summarizes these cases. In this table l and a denote respectively the length and the allocated size of string s, o denotes the offset of the pointer, c denotes the evaluated right-hand side of the assignment, and finally r denotes

sizeof(τ). The definitions of all these functions can be found in Appendix A.3 and they are similar to the definition of **before** in Figure 2. Using the 6 transformations aforementioned we can now define:

$$\mathbb{S}^\sharp[\![*_\tau \texttt{@[}s \in \mathfrak{V}\texttt{,}e_1\texttt{]=}e_2]\!](S^\sharp) =$$

- $\bigsqcup\{\mathbf{set0}(s, e_1', e_2', S^{\sharp\prime\prime}) \sqcup \mathbf{serror}(s, e_1', 1, S^{\sharp\prime\prime}) \sqcup \mathbf{unchanged}(s, e_1', e_2', S^{\sharp\prime\prime}) \sqcup \mathbf{setnon0}(s, e_1', e_2', S^{\sharp\prime\prime}) \mid (e_1', S^{\sharp\prime}) \in \mathbb{E}^\sharp[\![e_1]\!](S^\sharp), (e_2', S^{\sharp\prime\prime}) \in \mathbb{E}^\sharp[\![e_2]\!](S^{\sharp\prime})\}$ if $\tau =$ **u8**
- $\bigsqcup\{\mathbf{l\_unchanged}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime}) \sqcup \mathbf{forget}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime}) \sqcup \mathbf{serror}(s, e_1', \mathit{sizeof}(\tau), S^{\sharp\prime}) \mid (e_1', S^{\sharp\prime}) \in \mathbb{E}^\sharp[\![e_1]\!](S^\sharp)\}$ if $\tau \neq$ **u8**

Example 6. Going back to Example 5, we now assume that $R^\sharp$ is a numerical abstract state built from the constraint set: $\{\mathbf{t} < s_a, \mathbf{t} = s_l, \mathbf{p} + \langle \mathtt{u}, 2, \mathbf{s32}\rangle < s_l', s_l' < s_a'\}$. Moreover in the following $S^\sharp[E]$ denotes the abstract state $S^\sharp$ in which the numerical component has been extended with the constraints set E, and e denotes the expression $\mathbf{p} + \langle \mathtt{u}, 2, \mathbf{s32}\rangle$. $\mathbb{E}^\sharp[\![\texttt{*(@[}s'\texttt{,}e\texttt{])}]\!](S^\sharp) = \{([1; 255], S^\sharp[\{e \geq 0, e < s_l', e < s_a'\}]), (0, S^\sharp[\{e \geq 0, e = s_l', e < s_a'\}]), ([0; 255], S^\sharp[\{e \geq 0, e > s_l', e < s_a'\}])\}$. With a precise enough numerical domain (e.g. polyhedra), $S^\sharp[\{e \geq 0, e = s_l', e < s_a'\}]$, $S^\sharp[\{e \geq 0, e > s_l', e < s_a'\}]$ and $S^\sharp[\{e < 0 \vee e \geq s_a'\}]$ form empty partitions, meaning that in this example, they represent impossible cases. For similar reasons $\mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp)$ will compute abstract elements that are reduced to $\bot$ for functions **set0**, **unchanged** and **serror**. Therefore: $\mathbb{S}^\sharp[\![\texttt{stmt}]\!](S^\sharp) = \langle\{\mathbf{p}, \langle \mathtt{u}, 2, \mathbf{s32}\rangle, \mathbf{t}\}, R^{\sharp\prime}, \{\mathbf{p} \mapsto s', \mathbf{t} \mapsto s\}\rangle$ with $R^{\sharp\prime} = \{\mathbf{t} < s_a, \mathbf{t} + 1 \leq s_l, \mathbf{p} + \langle \mathtt{u}, 2, \mathbf{s32}\rangle < s_l', s_l' < s_a'\}$. This assignment made our abstraction lose the position of the first `'\0'` character, as it wrote a non-`'\0'` character in its place.

String declaration. When encountering a local variable declaration (**u8** `s[`n`]` with $n \in \mathbb{N}_{\geq 0}$ and $\mathtt{s} \in \mathfrak{V}$) we can set the allocated size of the string to n, and set the length to the range $[0, n]$ as shown in the following example: $\mathbb{S}^\sharp[\![\mathbf{u8}\ \texttt{s[27]}]\!](\langle\emptyset, \emptyset, \emptyset\rangle) = \langle\emptyset, \{s_l \geq 0, s_l \leq 27, s_a = 27\}, \emptyset\}\rangle$. The formal definition is straightforward (see Appendix A.4).

Example 7. Consider again Program 1.1 from the introductory example, analyzed starting from an abstract state $S^\sharp = \langle\{\mathbf{p}, \mathbf{q}\}, \{\mathbf{p} = 0, \mathbf{q} = 0, 0 \leq s_l < s_a, 0 \leq s_l' < s_a'\}, \{\mathbf{p} \mapsto s, \mathbf{q} \mapsto s'\}\rangle$. Note that the input state contains the information that $\mathbf{p}$ and $\mathbf{q}$ do not alias. The numerical invariant (the rest of the abstract state is not modified by the analysis) found at the beginning of line 2 is: $\{-\mathbf{p} + \mathbf{q} = 0, s_l' \geq \mathbf{p} + 1, s_a' - s_l' \geq 0, \mathbf{p} \geq 0, s_l \geq \mathbf{p}\}$. An **out_of_bounds** error is generated at line 3, indeed in the starting abstract state, no hypothesis is made on the relation between s_l' and s_a therefore there might be a buffer overrun at line 3. Finally the numerical invariant discovered at the end of line 6 is: $\{s_l' = s_l, \mathbf{q} = s_l, \mathbf{p} = s_l, s_a' \geq s_l + 1, s_l \geq 0, s_a \geq s_l + 1\}$, thus showing that we were able to infer that the two strings pointed to by `p` and `q` have the same size at the end of the analysis.

Dynamic memory allocation. As mentioned in Figure 1, we allow dynamic memory allocations. The Cell abstract domain as presented in Section 3 is not able to handle those. To model dyamic memory allocation, we consider a finite set $\mathcal{A}$ of heap addresses, derived from the allocation site using recency abstraction [2]: for each allocation site $\mathfrak{a}$, one abstract address, $\mathfrak{a}^s$, is used to model the last block allocated at $\mathfrak{a}$, and another one, $\mathfrak{a}^w$, to summarize the blocks allocated previously at $\mathfrak{a}$. While we perform weak updates on the later, we can perform strong updates on the former, which ensures a gain in precision.

```
void aux1(char** x,int e) {
  ●1*x = malloc(e);
}
void aux2(char** x,int e) {
  ●2*x = malloc(e);
}
int main() {
  char* x;
  aux1(&x,10); aux1(&x,20);
  aux1(&x,30); aux2(&x,40);
  *x = '\0';
}
```

Program 1.3: Dynamic memory allocation

Example 8. Consider now Program 1.3, and assume that $\mathfrak{a}^s$ and $\mathfrak{a}^w$ (resp. $\mathfrak{b}^s$ and $\mathfrak{b}^w$) are addresses for which we perform strong and weak update at program point ●1 (resp. ●2). Starting from $\top$ the analysis of the body of function `main`, we get: $\langle\{\mathbf{x}\}, \{0 \leq \mathfrak{a}_l^w \leq \mathfrak{a}_a^w, 10 \leq \mathfrak{a}_a^w \leq 20, 0 \leq \mathfrak{a}_l^s \leq \mathfrak{a}_a^s, \mathfrak{a}_a^s = 30, \mathfrak{b}_a^s = 40, \mathfrak{b}_l^s = 0, \mathbf{x} = 0\}, \{\mathbf{x} \mapsto \mathfrak{b}^s\}\rangle$ This state gives us that `x` points to a memory location starting from a `'\0'` character. We also note that information about the two first allocations made at program point $\mathfrak{a}$ have been collapsed into the $\mathfrak{a}^w$ address.

5 Modular analysis

In a `C` project that manipulates strings, calls to functions such as `strcpy`, `strcat` are performed many times and at many different call sites. Performing a modular analysis of such functions and inferring a summary that is reusable at subsequent calls has potential to greatly improve scalability. We chose to perform our modular analysis in a classic top-down fashion. This ensures that when a function is analyzed, we already have some information on its context (in particular, the possible pointer aliasing and variable range), which helps maintaining the precision of the function analysis. Therefore we would like to be able to infer a partial function that, given an input abstract state and a statement, can produce an output abstract state that is an over approximation of the abstract state we would have obtained by performing the analysis of the statement. Such a function will be called a *summary*. Note that substituting some of the statement analysis by a call to a summary is sound.

```
void strcat(char* dest, char* src)
{
  int i; int j;
  for (i=0; dest[i]!='\0';i++) ;
  for (j=0; src[j]!='\0';j++)
    dest[i+j] = src[j];
  dest[i+j] = '\0';
}
```

Program 1.4: `strcat`

Example 9. Given $\texttt{stmt} \in \textit{stmt}$, $(I^\sharp, O^\sharp) \in (\mathcal{S}_m^\sharp \times \mathcal{S}_m^\sharp)$ such that $\mathbb{S}^\sharp[\![\texttt{stmt}]\!](I^\sharp) \sqsubseteq O^\sharp$, let us define $\mathbf{R}^\sharp = \lambda(\texttt{stmt}', S^\sharp), \textbf{if } \texttt{stmt}' = \texttt{stmt} \wedge S^\sharp \sqsubseteq I^\sharp \textbf{ then } O^\sharp \textbf{ else } \texttt{undefined}$. $\mathbf{R}^\sharp$ is a summary function, built using an input/output relation. This can be easily generalized using a set of input/output relations. Moreover we can

remove constraints on $I^\sharp$ before the analysis of the body of the function in order to improve the reusability of the input/output relation obtained, the drawback being that the corresponding output abstract value will be greater thus losing precision. Furthermore computing and storing new $(I^\sharp, O^\sharp)$ relations whenever no existing summary could be used can cause the computation of input/output relations that will never be reused, hence the importance of generalizing $I^\sharp$ in the direction of newly discovered call contexts, so as to tailor summaries to actual call sites abstract values.

Remark 3. Consider the statement $\texttt{stmt} = \texttt{x = x + 1}$, and assume our abstract domain to be the interval domain [6]. For every input state of the form $\{x \mapsto [\alpha, \beta]\}$, the output state will be of the form $\{x \mapsto [\alpha + 1, \beta + 1]\}$. A summary function $\mathbf{R}^\sharp$ defined on $\{\texttt{stmt}\} \times \mathcal{S}_m^\sharp$ in the manner of Example 9 (with a finite list of input/output relations) will never yield an analyzer able to express that for every input $[\gamma, \delta]$ the output is $[\gamma + 1, \delta + 1]$. Indeed the interval domain would produce a set of input/output relations $\{[\alpha_i, \beta_i] \mapsto [\alpha_i + 1, \beta_i + 1]\}$, and for an input $[\gamma, \delta] \subsetneq [\alpha_0, \beta_0]$ we could only use as output abstract state $[\alpha_0 + 1, \beta_0 + 1]$, thus losing information compared to $[\gamma + 1, \delta + 1]$.

Using relational domains. Relational domains are able to express relations of the form $y = x + 1$. Such a relation can grasp the semantic of `stmt` from the previous remark. We use the relational aspect of the numerical domain to express relations not only between the values of variables, but also between their values and their input values. This idea was introduced in [7] and is also used in [11]. Consider two sets of variables $\mathcal{V} = \{x, y\}$ and $\mathcal{V}' = \{x', y'\}$ and the abstract element: $S^\sharp = \{x = y', y = x'\}$. Moreover assume at input that $\{x = 3, y = 5\}$, then using the meet provided by the numerical domain in order to instantiate $S^\sharp$ with input constraints: $\{x = 3, y = 5\}_{|\{x', y',\ x, y\}} \sqcap \{x = y', y = x'\} = \{x = 3, y = 5, x' = 5, y' = 3\}$, and finally $\{x = 3, y = 5, x' = 5, y' = 3\}_{|\{x,y\}} = \{x' = 5, y' = 3\}$. This example emphasized how relational domains are used to express precise input/output relations between numerical variables.

Building the summary function. We feel that two analyses starting from different aliasing patterns should be kept separated in order to improve precision. Indeed, analyzing `strcat(p,q)` (see Figure 1.4) without any hypothesis on the possible aliasing of `p` and `q` would result in a huge loss of precision and in false alarms being raised at every call (`p` and `q` might be aliased, which would raise a segmentation fault). Therefore we must use partitioning of the abstract domain, performing an analysis for every possible aliasing scheme would result in a combinatorial blow up, moreover we might perform analysis for partitions that will never occur at any call site. For these reasons we will only analyze partitions on demand. Our goal is therefore to build a summary that is a set of numerical relations such as defined above. The decision to extend a partition or to build a new one will be based on the "symbolic" part of the abstract domain. The heuristic we chose was to separate abstract states with different aliasing, but also those where the unification of the cell or string sets would induce major differences in the numerical domain set. Moreover

the summary function is extended on demand, meaning that when the analyzer encounters a function call, it tries to use an existing relation and if none can be found it builds a new relation or it generalizes an existing one. Generalization of a relation is done in the following way: assume known a relation with input $I^\sharp$ and an abstract state $S^\sharp$, such that $S^\sharp \not\sqsubseteq I^\sharp$. If the analyzer deems that $S^\sharp$ and $I^\sharp$ should be in the same relation (e.g. because they have the same aliasing), we perform a new analysis of the function starting from $I^\sharp \triangledown (S^\sharp \sqcup I^\sharp)$, that is a generalization, of $I^\sharp$, by the mean of the widening operator. This ensures that, given an aliasing, a function will be analyzed only a finite number of times and that the input of the obtained relation is tailored to the actual values at call site. Building numerical relations does not require a transformation of the intra-procedural iterator. Indeed variables are added to the numerical domain with equality constraints between primed and unprimed variables. Analysis is then performed as if primed variables were not present in the numerical domain and they are removed after storing the summary.

Example 10. Consider the statement `s32 x = a + 1` (with *typeof*(`a`) = **s32**), from input state: $\langle \{\mathbf{a}'\}, \{\mathbf{a}' = \mathbf{a}, \mathbf{a} \geq 0\}, \emptyset \rangle$. The output state is then $\langle \{\mathbf{a}', \mathbf{x}'\}, \{\mathbf{a}' = \mathbf{a}, \mathbf{x}' = \mathbf{a} + 1, \mathbf{a} \geq 0\}, \emptyset \rangle$. From this we deduce the relation: let $I^\sharp_\alpha$ be some input state, if the set of cells of $I^\sharp_\alpha$ is precisely $\{\mathbf{a}\}$ and if the numerical domain of $I^\sharp_\alpha$ satisfies the condition $\mathbf{a} = \alpha$ and if the pointer map is empty, then the best possible output state is $\langle \{\mathbf{a}, \mathbf{x}\}, \{\mathbf{a} = \alpha, \mathbf{x} = \alpha + 1\}, \emptyset \rangle$.

Example 11. Consider now the function `strcat` of Figure 1.4. The modular analysis of this function yields a relation stating that:

if `dest` points (at offset 0) to some memory location s, with length s_l and allocated size s_a, and if `src` points (at offset 0) to some memory location t with equivalent length and allocated size definition and $t \neq s$

then $\{s'_l = t_l + s_l, s'_a = s_a, t'_a = t_a, t'_l = t_l, t'_l \geq 0, t'_l \leq t'_a - 1, t'_l \leq s'_l, s'_l \leq s'_a - 1\}$

Therefore thanks to the $s'_l = t_l + s_l$ relation, if another call to `strcat` is performed in a state where $s_l = \alpha$ and $t_l = \beta$ for some α and β, our analyzer, can conclude (without reanalysis) that the length of the string t_l at the end of the analysis is $\alpha + \beta$.

Remark 4. The following improvements were added:

- in order to improve reusability, we increase the input state by removing some memory blocks (meaning we leave out constraints on these regions) from the input state. This plays the role of the framing rule in separation logic. Note that this improvement does not induce any precision loss.
- when a summary is created, some memory blocks are quantified universally, therefore when trying to apply a summary we try to unify the memory blocks from the actual input state with those of the summary input state.

6 Implementation

The analyzer was implemented in OCaml in the novel and still in development Mopsa framework. Mopsa enables a modular development of static analyzers

defined by abstract interpretation. An analyzer is built by choosing abstract domains, and combining them according to the user specification. Abstract domains are either predefined (e.g. Cell abstract domain, loop iterators, ...) or user-defined (e.g. String abstract domain). The String abstract domain was added to the library of existing domains, and a new inter-procedural iterator was added to implement the modular analysis presented in Section 5. The current analyzer is in development, it is able to analyze all `C` code fragments presented in this article, but can not tackle complete realist `C` projects yet. To test our modular string analysis, we thus considered the examples and benchmarks used in previous works on string analysis [1], [16].

In related works, Allamigeon et al. mentioned in [1], Section 5, that the most difficult example they had to deal with were calls to `strcpy` performed on string placed in a structure, itself placed in a matrix, and accessed via pointer manipulations (see Program 1.6, in Appendix B). This example was successfully analyzed with the version of `strcpy` defined in Program 1.1 and with an alternate implementation found in Qmail (see Program 1.7), the second case was more complex and required the use of partitioning. Our ability to easily deal with such manipulations comes from the use of the Cell domain to deal with low-level features of `C`.

We are able to tackle most of the programs from web2c mentioned in [16] (7 out of 9, programs that could not be analyzed are due to the fact that we do not have yet implemented all the features of the `C` language). The precision of this analysis (number of errors and false alarms) is similar to that of [16] and the execution time of the analyzer was always below 2 seconds. As an example consider Prog. 1.5, starting the analysis under the conditions that: `cp` points to `buf`, a buffer of size `BUFSIZ` before the first `'\0'` character produces alarms at line 9 and 10. Indeed under such hypothesis `strcpy` tries to write outside of `tbuf`. Note moreover that both [1] and [16] defined special abstract transformations for `strcpy`, whereas we perform a modular analysis of the function.

```
1  char * insert_long (cp)
2       char *cp;
3  {
4    char tbuf[BUFSIZ];
5    int i;
6    for (i=0;&buf[i]<cp;++i)
7      tbuf[i] = buf[i];
8    strcpy(&tbuf[i],"(long)");
9    strcpy(&tbuf[i + 6], cp);
10   strcpy(buf, tbuf);
11   return cp + 6;
12 }
```

Program 1.5: `insert_long` from web2c

7 Related works

Modular Static Analysis. Cousot and Cousot mentioned in [11] the importance of performing modular analyses and described several methods to design them. An efficient way is to use user-provided contracts as in [17]. Our goal was to infer contracts, as in [14], therefore works closest to ours would be the input/output inference performed by Bourdoncle in [4], however this method was limited to non-relational (interval) domains, unlike our method, which is thus more expressive (see Remark 3). In [11], numerical relations are used to represent the semantic of a set of statements, however this is limited to numerical programs whereas we extend the method to consider both numbers and pointers, including

pointer arithmetic. The analyzer proposed by Sotin and Jeannet in [23] is able to infer input/output relations of the form proposed in Section 5. Nevertheless they consider a subset of `C` that does not contain pointer arithmetic, union types nor pointer casts. Müller-Olm and Seidl [19] and Sharma and Reps [20] proposed domains specialized in the discovery of numerical input/output relations on statements, in both cases the relations discovery is performed during the analysis of the statement by a special domain. In Sec. 5 we mentioned that we implemented a mechanism to infer framing in order to improve analysis reusability, framing mechanisms are fundamentals in tools base on separation logic such as Smallfoot [3] or Infer [5].

String analysis. One popular technique to avoid buffer overflows is dynamic analysis. There is a long history of such technique (see [25] for some examples). These methods induce an overhead cost and do not prevent program failures. By contrast we employ static analysis. String are arrays of characters, therefore analysis methods proposed in [13] and [8] could be used to design static analyzers handling strings. The three following works are the closest to ours and all follow the idea introduced in [24] to track the length of strings. Dor et al. [16] tackled the problem by rewriting string manipulating statements into statements over a numerical variable language, however this transformation induced the usage of a number of variables quadratic in the number of strings present in the analysis, in order to account for pointer aliasing. Simon and King [22] proposed an analyzer for a sub-`C` language manipulating strings, and allowing dynamic memory allocation, but some pointer manipulations could not be handled. They improved their results in [21], the string domain presented here is a combination of results from [21] and the cell abstract domain, moreover we provided a way to dynamically balance strings dealt with by the string abstract domain and by the cell domain. Additionally string length and allocated size are bound to pointers (whereas we bind them to the actual memory location containing the string), and this approach seems to prevent the modular integration of this domain in a full `C` language analyzer. Allamigeon et al. [1] also proposed an analyzer that keeps track of the position of the first `'\0'` character, however their analysis is non-relational, can not handle arbitrary pointer cast, and uses static information on string length, therefore preventing the domain reusability for dynamically allocated strings. We believe that our analysis is the first one that is both modular, able to reason both on the C at a low-level (including pointer casts and unions), and at a higher-level (on strings using dedicated abstractions).

8 Conclusion

In this article we proposed an abstract domain able to tackle `C` string of parametric size, built as an add-on to an existing domain [18] capable of dealing with most of the features of the `C` language. We have shown how our analyzer can be tuned dynamically (by choosing whether the String or Cell domain should deal with certain memory regions, by changing the partitioning heuristics in the

inter-procedural iterator or by changing the underlying numerical domain) so as to adjust its precision. Upon the aforementioned analyzer we defined an inter-procedural iterator designed to increase statement analysis reusability without having to lose precision.

References

1. Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded C programs. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2006.
2. Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.
3. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
4. François Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Funct. Program.*, 2(4):407–423, 1992.
5. Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.
8. Patrick Cousot. Verification by abstract interpretation. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 243–268. Springer, 2003.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.
10. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

11. Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.
12. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.
13. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 105–118. ACM, 2011.
14. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 150–168. Springer, 2011.
15. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.
16. Nurit Dor, Michael Rodeh, and Shmuel Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2001.
17. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010.
18. Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In Mary Jane Irwin and Koen De Bosschere, editors, *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006*, pages 54–63. ACM, 2006.
19. Markus Müller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29(5):29, 2007.
20. Tushar Sharma and Thomas W. Reps. A new abstraction framework for affine transformers. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2017.

21. Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.
22. Axel Simon and Andy King. Analyzing string buffers in C. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2002.
23. Pascal Sotin and Bertrand Jeannet. Precise interprocedural analysis in the presence of pointers to the stack. In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 459–479. Springer, 2011.
24. David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*. The Internet Society, 2000.
25. John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

A Tool functions

A.1 Definition of the Galois connection

In the following st denotes $sizeof \circ typeof$.

$$
\begin{aligned}
&\textbf{to\_cell}(s, S^\sharp) = \\
&\qquad \textbf{let } \langle C, R^\sharp, P\rangle = \textbf{add\_cells}(\{\langle s, 0, \mathbf{u8}\rangle, \ldots, \langle s, st(s), \mathbf{u8}\rangle\}, S^\sharp) \textbf{ in} \\
&\qquad \textbf{let } [a; b] = \textbf{range}(s_l, R^\sharp) \cap [0; st(s)] \textbf{ in} \\
&\qquad \textbf{let } (R_i^\sharp)_{i\in[a;b]} = R^\sharp \sqcap \{\langle s, 0, \mathbf{u8}\rangle \neq 0,\ \ldots, \langle s, i-1, \mathbf{u8}\rangle \neq 0, \langle s, i, \mathbf{u8}\rangle = 0\} \textbf{ in} \\
&\qquad \bigsqcup_{j=0}^{b} \langle C', R_j^\sharp, P\rangle \\
&\textbf{from\_cell}(s, S^\sharp) = \\
&\qquad \textbf{let } \langle C, R^\sharp, P\rangle = \textbf{add\_cells}(\{\langle s, 0, \mathbf{u8}\rangle, \ldots, \langle s, st(s), \mathbf{u8}\rangle\}, S^\sharp) \textbf{ in} \\
&\qquad \textbf{let } c_\geq = \min(\{i \mid 0 \in \textbf{range}(\langle s, i, \mathbf{u8}\rangle, R^\sharp)\} \cup \{st(s)\}) \textbf{ in} \\
&\qquad \textbf{let } c_\leq = \min(\{i \mid \{0\} = \textbf{range}(\langle s, i, \mathbf{u8}\rangle, R^\sharp)\} \cup \{st(s)\}) \textbf{ in} \\
&\qquad \textbf{let } C^\star = \{\langle s', i, \tau\rangle \in C \mid s \neq s'\} \textbf{ in} \\
&\qquad \textbf{let } R^\star = R \sqcap \{s_l \geq c_\geq, s_l \leq c_\leq, s_a = st(s)\} \textbf{ in} \\
&\qquad \langle C^\star, R^\star, P\rangle
\end{aligned}
$$

Using functions **to_cells** and **from_cells**, and under the hypothesis that $\mathfrak{V} = \{s_0, \ldots, s_{n-1}\}$, we can define the Galois connection between the Cell abstract

domain and the String abstract domain:

$$\gamma_{S^\sharp_m, D^\sharp_m}(S^\sharp) = \textbf{to\_cell}(s_0, \ldots, (\textbf{to\_cell}(s_{n-1}, S^\sharp)) \ldots)$$
$$\alpha_{S^\sharp_m, D^\sharp_m}(S^\sharp) = \textbf{from\_cell}(s_0, \ldots, (\textbf{from\_cell}(s_{n-1}, S^\sharp)) \ldots)$$

A.2 Definition of the evaluation of a dereferencing

$$\begin{aligned}
\textbf{before}(e, s, S^\sharp) = & \\
& \{([1; 255], \langle C, R^\sharp \sqcap \{0 \leq e, e < s_l, e < s_a\}, P\rangle\} \\
\textbf{at}(e, s, S^\sharp) = & \\
& \{(0, \langle C, R^\sharp \sqcap \{0 \leq e, e = s_l, e < s_a\}, P\rangle\} \\
\textbf{after}(e, s, S^\sharp) = & \\
& \{([0; 255], \langle C, R^\sharp \sqcap \{0 \leq e, e > s_l, e < s_a\}, P\rangle\} \\
\textbf{eerror}(e, s, S^\sharp) = & \\
& \textbf{let } R^\sharp_1 = R^\sharp \sqcap \{e \geq s_a\} \sqcup R^\sharp \sqcap \{e < 0\} \textbf{ in} \\
& \textit{test for } \textbf{\textit{out_of_bounds}} \ (R^\sharp_1 \sqsubseteq_{S^\sharp_m} \bot)? \\
& \emptyset
\end{aligned}$$

A.3 Definition of the abstract postcondition of an assignment

$$\begin{aligned}
\textbf{set0}(s, e_1, e_2, \langle C, R^\sharp, P\rangle) = & \\
& \textbf{let } R^\sharp_1 = R^\sharp \sqcap \{e_1 \geq 0, e_1 \leq s_l, e_1 < s_a, e_2 = 0\} \textbf{ in} \\
& \textbf{let } R^\sharp_2 = \mathbb{S}^\sharp[\![s_l \leftarrow e_1]\!](R^\sharp_1) \textbf{ in} \\
& \langle C, R^\sharp_2, P\rangle \\
\textbf{setnon0}(s, e_1, e_2, \langle C, R^\sharp, P\rangle) = & \\
& \textbf{let } R^\sharp_1 = R^\sharp \sqcap \{e_1 \geq 0, e_1 = s_l, e_1 < s_a, e_2 \neq 0\} \textbf{ in} \\
& \textbf{let } R^\sharp_2 = \mathbb{S}^\sharp[\![s_l \leftarrow [e_1 + 1; s_a]]\!](R^\sharp_1) \textbf{ in} \\
& \langle C, R^\sharp_2, P\rangle \\
\textbf{unchanged}(s, e_1, e_2, \langle C, R^\sharp, P\rangle) = & \\
& \textbf{let } R^\sharp_1 = (R^\sharp \sqcap \{e_1 \geq 0, e_1 < s_l, e_1 < s_a, e_2 \neq 0\}) \\
& \qquad \sqcup (R^\sharp \sqcap \{e_1 \geq 0, e_1 > s_l, e_1 < s_a\}) \textbf{ in} \\
& \langle C, R^\sharp_1, P\rangle
\end{aligned}$$

$$
\begin{aligned}
&\textbf{l\_unchanged}(s, e_1, r, \langle C, R^\sharp, P\rangle) = \\
&\qquad \textbf{let } R_1^\sharp = (R^\sharp \sqcap \{e_1 \geq 0, e_1 > s_l, e_1 + r \leq s_a\}) \textbf{ in} \\
&\qquad \langle C, R_1^\sharp, P\rangle \\
&\textbf{forget}(s, e_1, r, \langle C, R^\sharp, P\rangle) = \\
&\qquad \textbf{let } R_1^\sharp = R^\sharp \sqcap \{e_1 \geq 0, e_1 \leq s_l, e_1 + r \leq s_a\} \textbf{ in} \\
&\qquad \textbf{let } R_2^\sharp = \mathbb{S}^\sharp[\![s_l \leftarrow [e_1; s_a]]\!](R_1^\sharp) \textbf{in} \\
&\qquad \langle C, R_2^\sharp, P\rangle \\
&\textbf{serror}(s, e_1, r, \langle C, R^\sharp, P\rangle) = \\
&\qquad \textbf{let } R_1^\sharp = R^\sharp \sqcap \{e_1 \geq s_a\} \textbf{ in} \\
&\qquad \textbf{let } R_2^\sharp = R^\sharp \sqcap \{e_1 < 0\} \textbf{ in} \\
&\qquad \textit{test for } \textbf{\textit{out\_of\_bounds}}\ ((R_2^\sharp \sqcup_{S_m^\sharp} R_1^\sharp) \sqsubseteq_{S_m^\sharp} \bot)\ ? \\
&\qquad \bot
\end{aligned}
$$

A.4 String declaration

$$\mathbb{S}^\sharp[\![\textbf{u8}\ \ \mathtt{s} \in \mathfrak{V}[n \in \mathbb{N}_{\geq 0}]]\!](\langle C, N^\sharp, P\rangle) = \langle C, \mathbb{S}^\sharp[\![\mathtt{s}_l \leftarrow [0, n]]\!](\mathbb{S}^\sharp[\![\mathtt{s}_a \leftarrow n]\!](N^\sharp)), P\rangle$$

B C programs

```
typedef struct {
  char* f;
} s;
char buf[10];

void init(s* x) {
  x[1].f = buf;
}
int main () {
  s a[2][2];
  s* ptr = (s*) &(a[1]);
  init(ptr);
  ptr = (s*) &(a[0]);
  strcpy(a[1][1].f,"strcpy ok");
  strcpy(a[1][1].f,"strcpy not ok");
}
```

Program 1.6: Program from [1]

```
void strcpy(char* s, char* t)
{
  for (;;) {
    if (!(*s = *t)) return ; ++s; ++t;
    if (!(*s = *t)) return ; ++s; ++t;
    if (!(*s = *t)) return ; ++s; ++t;
    if (!(*s = *t)) return ; ++s; ++t;
  }
}
```

Program 1.7: Strcpy from Qmail