String abstraction for model checking of C programs*

Agostino Cortesi[1], Henrich Lauko[2], Martina Olliaro[1], Petr Ročkai[2]

[1] Ca' Foscari University, Via Torino 155, Venezia Mestre, Italy,
cortesi@unive.it, martina.olliaro@unive.it
[2] Faculty of Informatics, Masaryk Universit, Brno, Czech Republic,
xlauko@mail.muni.cz, xrockai@fi.muni.cz

Abstract. Automatic abstraction is a powerful software verification technique. In this paper, we elaborate an abstract domain for C strings, that is, null-terminated arrays of characters. We describe the abstract semantics of basic string operations and prove their soundness with regards to previously established concrete semantics of those operations. In addition to a selection of string functions from the standard C library, we provide semantics for character access and update, enabling automatic lifting of arbitrary string-manipulating code into the domain.
The domain we present (called M-String) has two other abstract domains as its parameters: an index (bound) domain and a character domain. Picking different constituent domains allows M-String to be tailored for specific verification tasks, balancing precision against complexity.
In addition to describing the domain theoretically, we also provide an executable implementation of the abstract operations. Using a tool which automatically lifts existing programs into the M-String domain along with an explicit-state model checker, we have evaluated the proposed domain experimentally on a few simple but realistic test programs.

1 Introduction

The C programming language is still very relevant [3]: a large number of systems of critical importance are written in C, including server software and embedded systems. Unfortunately, due to the way C programs are laid out in memory, they often contain bugs that can be exploited by malicious parties to mount security attacks. Guaranteeing correctness of such software is of great concern. In particular, we are interested in ensuring correctness of C programs that manipulate strings. Incorrect string manipulation can cause a number of catastrophic events, ranging from crashes in critical software components to loss or exposure of sensitive data.

In the C programming language, strings are not a basic data type and operations on them are provided as library functions [7]. Indeed strings are represented as zero-terminated arrays of characters – due to the possible discrepancy between string size and array (buffer) size, C programs which manipulate strings can suffer from buffer overflows and related issues. A buffer overflow is a bug that affects C code which incorrectly tries to access a buffer outside its bounds – an out-of-bounds write (a related bug – an out-of-bounds read – is also a problem, even though not as immediately dangerous as a buffer

* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

overflow). Moreover, buffer overflows are usually exploitable and often can easily lead to arbitrary code execution [25]. In the light of these facts, it is clearly important to investigate methods to automatically reason about correctness of string manipulation code in C programs. Automated code analysis tools can identify existing bugs, reduce the risk of introducing new bugs and therefore help prevent costly security incidents.

In this paper, we present a sound approach for conducting string analysis in C programs. In particular, we consider the M-String segmentation abstract domain [10]. We use it to perform abstraction-based model checking [9] of C programs, with focus on string manipulation. The model checker is split into two parts, as proposed in [23]: a program transformation which changes the program to execute in the abstract domain, and a standard, explicit-state model checker which exhaustively explores the abstract state space.

1.1 Related work

Static methods with the ability to automatically detect buffer overflows have been widely studied in the literature and many different inference techniques were proposed and implemented: constraint solvers for various theories (including string theories) and techniques based on them (e.g. symbolic execution), tainted data-flow analysis, string pattern matching analysis or annotation analysis [27]. Additionally, a large number of bug hunting tools based on static analysis and the above mentioned techniques have been implemented [1, 14, 16, 17, 29, 30].

For instance, in [19] authors introduced a performant backward compatible method of bounds checking of C program, i.e., the representation of pointers is left unchanged (thus differentiating the proposed schema from previously existing techniques), allowing inter-operation between checked and unchecked code, with recompilation confined to the modules where problems might occur. In [14], a static verifier of C strings has been presented, namely CSSV. Contracts are supplied to the tool, which acts in 4 stages, reducing the problem of checking code that manipulates string to checking code that manipulates integers. Finally, Splat, described in [31], is a tool that automatically generates test inputs, symbolically reasoning about lengths of input buffers.

Briefly, static code analysis attempts to quickly approximate possible behaviours of a program, without examining its actual executions. This way, static analysis reasons about many of the possible runs of a program and provides a degree of assurance that the property of interest holds (or that it is violated). However, with static analysis, neither positive nor negative results are guaranteed to be correct [2].

To obtain a higher degree of confidence, a number of more expensive methods are available in the software verification toolbox [15]. Model checking with abstraction and refinement is one such high-assurance, high-precision method [9], though of course both the precision and reliability come at a price in terms of computational complexity.

Various researchers have shown how the framework of abstract interpretation [12] can be used to approximate semantics of string operations. The basic, well-known domains are a *string set* domain, which simply keeps track of a set of strings – this is specific instance of the general (bounded) set domain. Another is the *character inclusion* domain (which keeps track of which characters appear in a string, but not in what order or how many times), the *prefix-suffix* domain (which keeps track of the first and the last

letter) and their various products. Another general-purpose string domain is the *string hash* domain proposed in [24], based on a distributive hash function. A more complete review of general-purpose string domains is readily available in the literature, e.g. [5, 11].

Such general-purpose domains focus on the generic aspects of strings, without accounting for the specifics of string handling in different programming languages. It is, however, often beneficial to consider such specific aspects of string representation when designing abstract domains for program analysis: indeed, M-String is a domain tailored specifically for the representation of strings used in C programs. A number of abstract string domains (and their combinations) for analysis of JavaScript programs have been evaluated in [5]. Another domain that was conceived for JavaScript analysis is the simplified regular expression domain defined in [26]. While dynamic languages heavily rely on strings and their analysis benefits greatly from tailored abstract domains, the specifics of the C approach to strings also deserves attention: the M-String domain, tailored for modeling zero-terminated strings stored in character buffers in C programs has first been described in [10]. In addition to theoretical work, a number of tools based on the abovementioned abstract domains and their combinations have been designed and implemented [18, 20, 26, 28].

Finally, combining many domains in a single analysis can often substantially improve precision over either of the individual domains. However, combining domains naively requires a quadratic number of translation functions. A solution to this problem, with special focus on string domains, has been proposed in [4]. Moreover, analysis of strings based on abstract interpretation is not limited to designing abstract string domains – an analysis for programs which process structured text, based on grammar inference, was proposed in [21]. A related approach based on over-approximation of string expressions using regular grammars (widened from context-free grammars constructed via static analysis) is described in [8].

1.2 Paper contribution

In this paper we define the semantics of the M-String abstract domain, based on the concrete semantics presented in [10], both in human-readable and in executable form. Additionally, we have extended LART [23], a tool which can perform automatic abstraction on programs, with support for more complicated (non-scalar) domains, which allowed us to also integrate the M-String domain. By using the extended version of LART along with DIVINE 4 [6], an explicit state model checker based on LLVM, we can automatically verify correctness of string operations in C programs. We demonstrate this capability by analysing a number of C programs, ranging from quite simple to moderately complex, including parsers generated by `bison`, a tool which translates context-free grammars into C parsers. The main contribution of this paper is in demonstrating the actual impact of an ad-hoc segmentation-based abstract domain on model checking of C programs.

2 M-String

M-String (**M**) [10] is an ad hoc segmentation-based abstract domain designed for string analysis in C programs, based on a refinement of the segmentation approach to array

representation proposed in [13]. In [13], the array's content is abstracted by consecutive, non-overlapping segments covering all array elements. In [10] the authors took advantage of this representation and defined a domain that abstracts C-like strings, distinguishing the so-called *string of interest*[3] of a character array from the rest of its content.

The goal of the domain is to infer the presence of common string manipulation errors that may result in buffer overflows or, more generally, that may lead to undefined behaviours. Additionally, keeping track of the content of the char array after the first null character allows us to reduce false positives: in particular, rewriting the first null character in the string is not always a bug, since further null characters may follow. Finally, M-String, like the array segmentation-based representation defined in [13], is parametric with respect to the abstraction of the array elements value, and the representation of array indices.

2.1 Concrete domain

Let A be a finite set of characters representable by the character encoding in use and let $\mathbb{C} = \mathsf{A}^*$ be the set of all the possible character arrays. Then, the operational semantics of character array variables ($\mathtt{c} \in \mathbb{C}$) are concrete array environments $\mu \in \mathcal{R}_m$ mapping character array names $\mathtt{c} \in \mathbb{C}$ to their values $\mu(\mathtt{c}) \in \mathcal{M} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times M \times \mathbb{Z}$, where:

- $\mathcal{R}_v \triangleq \mathbb{X} \to \mathcal{X}$ is the environment which maps names $\mathtt{x} \in \mathbb{X}$ to values $\rho(\mathtt{x}) \in \mathcal{X}$,
- $\mathbb{E}$ is the expressions domain,
- $M : \mathbb{Z} \to \mathbb{Z} \times \mathsf{A}$, and $\mathbb{Z}$ is the integers domain.

For more details we invite the reader to refer to [10, 13]. Moreover, we highlight the fact that the concrete domain we present is used as a framework that helps us in constructing the abstract representation, and it is not how the (concrete) values are actually represented in programs. That said, let $\mathtt{c}$ be an array of characters. Its concrete value is a quintuple $\mu(\mathtt{c}) = (\rho, \mathtt{c.low}, \mathtt{c.high}, M_\mathtt{c}, \mathrm{N}_\mathtt{c}) \in \mathcal{M}$ where $\rho \in \mathcal{R}_v$ and:

- $\mathtt{c.low}, \mathtt{c.high} \in \mathbb{E}$ are expressions whose values $[\![\mathtt{c.low}]\!]\rho$ and $[\![\mathtt{c.high}]\!]\rho$ respectively represent the integer lower bound and the integer upper bound of $\mathtt{c}$,
- $M_\mathtt{c}$ is a function that maps an index i to a pair $M_\mathtt{c}(\mathrm{i}) = \langle \mathrm{i}, v \rangle$ of the index i and the corresponding character array element value v, i.e. $M_\mathtt{c} : \mathrm{I}_\mathtt{c} \to \mathrm{P}_\mathtt{c}$ such that:

$$\mathrm{I}_\mathtt{c} = \{\mathrm{i} : \mathrm{i} \in [[\![\mathtt{c.low}]\!]\rho, [\![\mathtt{c.high}]\!]\rho)\}$$
$$\mathrm{P}_\mathtt{c} = \{\langle \mathrm{i}, v \rangle : \mathrm{i} \in [[\![\mathtt{c.low}]\!]\rho, [\![\mathtt{c.high}]\!]\rho \wedge \mathtt{c}[\mathrm{i}] = \text{'}v\text{'}\}$$

- $\mathrm{N}_\mathtt{c}$ is the set of indexes which map to the string terminating characters, i.e. $\mathrm{N}_\mathtt{c} = \{\mathrm{i} \in [[\![\mathtt{c.low}]\!]\rho, [\![\mathtt{c.high}]\!]\rho) \mid M_\mathtt{c} = \langle \mathrm{i}, \text{'}\backslash 0\text{'} \rangle\}$.

Example 1. Let $\mathtt{s} =$ "Hello\0" be a character array then, its concrete value is given by $\mu(\mathtt{s}) = (\rho, 0, 6, M_\mathtt{s}, \mathrm{N}_\mathtt{s})$, where $\mathrm{P}_\mathtt{s}$ is the set $\{(0, \text{'H'}), (1, \text{'e'}), (2, \text{'l'}), (3, \text{'l'}), (4, \text{'o'}), (5, \text{'\0'})\}$ and $\mathrm{N}_\mathtt{s}$ corresponds to the singleton $\{5\}$.

[3] The *string of interest* of a character array is the sequence of characters up to the first null one (included). In the case in which the null character occurs at the first index of a character array, then its *string of interest* is defined as "null". If the null character does not occur in the array, then its *string of interest* is defined as "undefined". Otherwise, the *string of interest* is considered to be "well-defined".

2.2 Abstract domain

The M-String (**M**) abstract domain approximates sets of character arrays with a pair of segmentations that highlight the nature of their *strings of interest*. The elements of the domain are split segmentation abstract predicates. Segments capture sequences of identical abstract values, and are delimited by so-called segment bounds. More precisely, the M-String abstract domain is given by **M**(**B**,**C**,**R**). **R** denotes the abstraction of scalar variable environments. **C** is the abstraction of the character array elements, and it is equipped with `is_null`, a special monotonic function lifting abstract elements in **C** to a value in the set {`true`, `false`, `maybe`}. **B** denotes the abstraction of segment bounds, equipped with the following operations: equality ($=_{\mathbf{B}}$), ordering ($\leqslant_{\mathbf{B}}$), least upper bound between subsequent segment bounds ($\sqcup_{\mathbf{B}}[b_i, b_{i+1})$), addition ($+_{\mathbf{B}}$), and subtraction ($-_{\mathbf{B}}$). The M-String abstract domain is the complete lattice $(\overline{\mathcal{M}}, \leqslant_{\mathbf{M}}, \bot_{\mathbf{M}}, \top_{\mathbf{M}}, \sqcap_{\mathbf{M}}, \sqcup_{\mathbf{M}})$ where:

- $\overline{\mathcal{M}} \triangleq (\overline{\mathcal{M}}_s, \overline{\mathcal{M}}_{ns}) \cup \{\bot_{\mathbf{M}}, \top_{\mathbf{M}}\}$
  - $\overline{\mathcal{M}}_s$ corresponds to $\bigcup\{\overline{\mathcal{S}}_{sb} \times \overline{\mathcal{S}}_{sm}^k \times \overline{\mathcal{S}}_{se} \mid k \geqslant 0\} \cup \overline{\mathcal{S}}_{se} \cup \{\emptyset\}$, and it represents the segmentation of the *string of interest* of a given character array, where,
  $$[\overline{\mathcal{S}}_{sb} = \{\overline{\mathcal{B}} \times \overline{\mathcal{C}}\}, \overline{\mathcal{S}}_{sm} = \{\overline{\mathcal{B}} \times \overline{\mathcal{C}} \times \{\text{␣}, ?\}\}, \overline{\mathcal{S}}_{se} = \{\overline{\mathcal{B}} \times \{\text{␣}\}\}]$$
  - $\overline{\mathcal{M}}_{ns}$ corresponds to $\bigcup\{\overline{\mathcal{S}}_{nsb} \times \overline{\mathcal{S}}_{nsm}^k \times \overline{\mathcal{S}}_{nse} \mid k \geqslant 0\} \cup \{\emptyset\}$, and it represents the segmentation of the content of a given character array after its *string of interest*, or character arrays that do not contain the null terminating character. Here,
  $$[\overline{\mathcal{S}}_{nsb} = \overline{\mathcal{S}}_{sb}, \overline{\mathcal{S}}_{nsm} = \overline{\mathcal{S}}_{sm}, \overline{\mathcal{S}}_{nse} = \{\overline{\mathcal{B}} \times \{\text{␣}, ?\}\}]$$

  In particular:
  1. $b_i \in \overline{\mathcal{B}}$ denotes the segment bounds, such that $i = 1, \ldots, n$ and $n > 1$ (notice that b_1 and b_n respectively represent the array lower bound and the array upper bound),
  2. $p_i \in \overline{\mathcal{C}}$ are abstract predicates, chosen in an abstract domain **C**, denoting possible values of pairs $\langle \mathrm{i}, v \rangle$ in a segment (i.e. $\mathbf{C}[\![\langle \mathrm{i}, v \rangle]\!]\overline{\rho}$),
  3. the question mark ? indicates the preceding segment might be empty, while ␣ indicates a non-empty segment

  The elements in $\overline{\mathcal{M}}$ are $\overline{m} = (s, ns)$ (i.e. split segmentation abstract predicates). Let $\mathtt{c} \in \mathbb{C}$ be an array of characters, and $\mu(\mathtt{c})$ be its concrete value; for instance, if the *string of interest* of c is null (i.e. $min(\mathrm{N}_{\mathtt{c}}) = 0$) then: $\overline{m}$ is equal to $(b_1 \text{␣}, \emptyset)$ if the size of c is equal to 1, $(b_1 \text{␣}, b_2 p_2 b_3 ?^3 p_3 b_4 ?^4 \ldots b_n ?^n)$ otherwise. In the rest of the paper we will refer to the *s* and to the *ns* parameters of a given abstract string $\overline{m}$ by $\overline{m}.s$ and $\overline{m}.ns$ respectively.
- Let $\overline{m}_1$ and $\overline{m}_2$ be two abstract values in the M-String domain then: $\overline{m}_1 \leqslant_{\mathbf{M}} \overline{m}_2 \Leftrightarrow \overline{m}_1 = \bot_{\mathbf{M}} \vee \overline{m}_1 \equiv \overline{m}_2 \vee \mathrm{unify}(\overline{m}_1, \overline{m}_2) = \overline{m}_2$. Notice that $\overline{m}_1$ and $\overline{m}_2$ are equivalent when they represent the same set of character arrays. Here, "unify" is a sound upper bound operator (originally defined in [13] and tweaked in [10] to modify two split segmentations so that they coincide).
  Take $\overline{m}_1$ and $\overline{m}_2$ to be compatible if their parameters have common lower and upper bounds of *s* and *ns*. Then, $\mathrm{unify}(\overline{m}_1, \overline{m}_2) = (\mathrm{unify}(s_1, s_2), \mathrm{unify}(ns_1, ns_2))$ if $\overline{m}_1$ and $\overline{m}_2$ are compatible, $\top_{\mathbf{M}}$ otherwise.
- $\bot_{\mathbf{M}}$, $\top_{\mathbf{M}}$ are special elements denoting the bottom/top element of the lattice.

– $\sqcup_{\mathbf{M}}$ represents the join operator, that defines the least upper bound between two abstract elements, such that: $\overline{m}_1 \sqcup_{\mathbf{M}} \overline{m}_2 = \text{unify}(\overline{m}_1, \overline{m}_2)$ if $\overline{m}_1$ and $\overline{m}_2$ are compatible, $\top_{\mathbf{M}}$ otherwise. Then the character abstract domain join is applied segment-wise.

Abstraction Let X be a set of concrete character array values. The abstraction function on the M-String abstract domain $\alpha_{\mathbf{M}}$ maps X to $\bot_{\mathbf{M}}$ in the case in which X is empty, otherwise to the pair of segmentations that best over-approximate values in X.

Concretization The concretization function on the M-String abstract domain $\gamma_{\mathbf{M}}$ maps an abstract element to a set of character arrays values as follows: $\gamma_{\mathbf{M}}(\bot_{\mathbf{M}}) = \emptyset$, otherwise $\gamma_{\mathbf{M}}(\overline{m})$ is the set of all possible character arrays values represented by a split segmentation abstract predicate $\overline{m}$. The formalization is quite complex, and the reader may refer to Appendix A.1.

Example 2. Let $\mathtt{S} = \{\mathtt{s}_1, \mathtt{s}_2, \mathtt{s}_3\}$ be a set of character arrays, such that: $\mathtt{s}_1$ ="`car\0xx`", $\mathtt{s}_2$ ="`bay\0xx`", and $\mathtt{s}_3$ ="`day\0xx`". The abstract value of $\mathtt{S}$ in $\mathbf{M}$, instantiated with the standard constant propagation domain ($\mathcal{CP}$), is given by $\overline{s} = \alpha_{\mathbf{M}}(\{\mu(\mathtt{x}) \mid \mathtt{x} \in \mathtt{S}\}) = (\{0\}\ \top_{\mathcal{CP}}\ \{1\}\ \text{'a'}\ \{2\}\ \top_{\mathcal{CP}}\ \{3\}, \{4\}\ \text{'x'}\ \{6\})$. The concretization function of $\overline{s}$, i.e. $\gamma_{\mathbf{M}}(\overline{s})$ maps $\overline{s}$ to the set of all possible character arrays values of length 6 that contain a *string of interest* of length 4, and having the character 'a' at position 1 and the character 'x' at position 4 and 5.

2.3 Abstract semantics

In [10] authors restricted their focus on a small representative set of operators which are part of the `string.h` library of the C programming language (i.e. `strcpy`, `strcat`, `strlen`, `strchr`, `strcmp` and the "assignment to an array element" operator), and they defined the concrete semantics of those operators. We recall the character arrays concrete semantics (slightly modified from the one presented in [10]). In particular, $\mathfrak{S}$ is the semantics that, given a statement and eventually some concrete character arrays values in $\mathcal{M}$, returns a concrete character array resulting from that operation, i.e. $\mathfrak{S} : \mathit{Stm} \times \mathcal{M} \to \mathcal{M} \cup \{\mathsf{null}\}$ where, null denotes unknown values. Moreover, for `strlen` and `strcmp` we give the semantics $\mathfrak{L} : \mathit{Stm} \times \mathcal{M} \to \mathbb{Z} \cup \{\top_{\mathbb{Z}}\}$.

Below we present the abstract semantics of the `strcat`, `strlen` and `strchr` operators, and we prove their soundness (the reader interested in the definitions of the abstract semantics and of the proofs of soundness of the complete set of operators introduced above may refer to Appendix A.4). We denote by $\mathfrak{S}_{\mathbf{M}}$ and $\mathfrak{L}_{\mathbf{M}}$ the abstract counterparts of $\mathfrak{S}$ and $\mathfrak{L}$ respectively, such that: $\mathfrak{S}_{\mathbf{M}} : \mathit{Stm} \times \mathbf{M} \to \mathbf{M}$ and $\mathfrak{L}_{\mathbf{M}} : \mathit{Stm} \times \mathbf{M} \to \mathbf{B}$.

Additional operators We present some additional abstract operators useful to define the abstract semantics. Their complete algorithms are defined in Appendix A.3.

Length operators (minLen, maxLen, Len) We introduce the notions of minimum and maximum length of a split segmentation abstract predicate, and the length of the *strings of interest* that it represents. Precisely, we define: $\text{minLen}(\overline{m}) = \mathit{min}\{\mathit{len}(x) \mid x \in \gamma_{\mathbf{M}}(\overline{m})\}$,

$\text{maxLen}(\overline{m}) = max\{len(x) \mid x \in \gamma_{\mathbf{M}}(\overline{m})\}$, and $\text{Len}_{\overline{m}.s} = max\{len(x) \mid x \in \gamma^{\star}_{\mathbf{M}}(\overline{m}.s)\}$ (see Appendix A.1 for the definition of $\gamma^{\star}_{\mathbf{M}}$), where $len(x)$ denotes the size of a concrete character array value.

Segment concatenation ($\oplus$) Let bpb' and uru' be two segments ($b, b', u, u' \in \overline{\mathcal{B}}$ and $p, r \in \overline{\mathcal{C}}$) then, their concatenation is defined as follows: $bpb' \oplus uru'$ such that $bpb' \oplus uru' = bpb'ru^*$ where $u^* = b' + (u' - u)$. In the case in which the right hand side operand is a segmentation, then all the segment bounds belonging to it are modified accordingly. Question marks, if present, are left unchanged.

Abstract semantics of `strcat`: If the input ($\overline{m}_1$ and $\overline{m}_2$ respectively) approximate character arrays that contain a well-defined or null *string of interest*, and if the minLen of $\overline{m}_1$ is greater than or equal to the the sum of $\text{Len}_{\overline{m}_1.s}$ and $\text{Len}_{\overline{m}_2.s}$ minus one then the `strcat` returns $\overline{m}'_1$ where $\overline{m}_2.s$ has been appended to $\overline{m}_1.s$ and the following segments are modified accordingly. Otherwise it returns $\top_{\mathbf{M}}$.

Let $\overline{m}.s[b_i]^{\text{s}}$ ($\overline{m}.ns[b_i]^{\text{s}}$) be the left hand side parameter (the right hand side parameter) of $\overline{m}$ starting from the i-th segment bound. Conversely, $\overline{m}.s[b_i]_{\text{u}}$ ($\overline{m}.ns[b_i]_{\text{u}}$) is the left hand side parameter (the right hand side parameter) of $\overline{m}$ up to the i-th segment bound. Then $\mathfrak{S}_{\mathbf{M}}[\![\texttt{strcat}]\!](\overline{m}_1, \overline{m}_2)$ is equal to:

- $\overline{m}'_1$ if $\overline{m}_1 \neq (\emptyset, ns)$, $\overline{m}_2 \neq (\emptyset, ns)$ and $\text{minLen}(\overline{m}_1) \geqslant (\text{Len}_{\overline{m}_1.s} + \text{Len}_{\overline{m}_2.s} - 1)$;
- $\top_{\mathbf{M}}$ otherwise, where:

1. If $\overline{m}_1 = (b_1 p_1 ... b_{i\sqcup}, ns) \wedge \overline{m}_2 = (b_{1\sqcup}, ns) \Rightarrow \overline{m}'_1 = (\overline{m}_1.s[b_i]_{\text{u}} \oplus \overline{m}_2.s, \overline{m}_1.ns)$
2. If $\overline{m}_1 = (b_1 p_1 ... b_{i\sqcup}, ns) \wedge \overline{m}_2 = (b_1 p_1 ... b_{i\sqcup}, ns) \Rightarrow \overline{m}'_1 = (\overline{m}_1.s[b_i]_{\text{u}} \oplus \overline{m}_2.s, \overline{m}_1.ns[b^*]^{\text{s}})$, where $b^* = (b_{i_{\overline{m}_2}} -_{\mathbf{B}} b_{1_{\overline{m}_2}}) +_{\mathbf{B}} b_{i_{\overline{m}_1}} +_{\mathbf{B}} 1$

Notice that, question marks, if present, are left unchanged.

Abstract semantics of `strlen`: If the input split segmentation abstract predicate ($\overline{m}$) approximates character arrays that contain a well-defined or null *string of interest* then the `strlen` operator returns the least upper bound between the segment bounds which limit a certainly or maybe `is_null` segment abstract predicate. Otherwise it returns $\top_{\mathbf{B}}$.

Let $\overline{\text{x}}$ be an abstract character value (i.e. $\mathbf{C}[\![v]\!]\overline{\rho}$) appearing in a generic segment abstract predicate p. Formally, $\mathfrak{L}_{\mathbf{M}}[\![\texttt{strlen}]\!](\overline{m})$ is equal to:

- $\bigsqcup_{\mathbf{B}}_{\forall b_i \in \overline{m}.s} \{\sqcup_{\mathbf{B}}[b_i, b_{i+1}) \mid \overline{\text{x}} \text{ occurs in } p_i \wedge \overline{\text{x}} \textit{ may be null}\}$ if $\overline{m} \neq (\emptyset, ns)$;
- $\top_{\mathbf{B}}$ otherwise, where $\sqcup_{\mathbf{B}}[b_i, b_{i+1})$ is a shorthand for $b_i \sqcup_{\mathbf{B}} b_i + 1 \sqcup_{\mathbf{B}} b_i + 2 \sqcup_{\mathbf{B}} ... \sqcup_{\mathbf{B}} b_{i+1} - 1$, and it returns the set of elements in the interval $[b_i, b_{i+1})$.

Abstract semantic of `strchr`: If the input split segmentation abstract predicate ($\overline{m}$) approximates character arrays that contain a well-defined or null *string of interest*, and if the abstract character we are looking for ($\overline{\text{x}} \in \mathbf{C}$) appears in $\overline{m}.s$ then the $\texttt{strchr}_{\overline{\text{x}}}$ operator returns a split segmentation abstract predicate denoting the sub-segmentation of its left hand side input parameter starting from the first occurrence of $\overline{\text{x}}$. Otherwise it returns $\top_{\mathbf{M}}$. Formally, $\mathfrak{S}[\![\texttt{strchr}_{\overline{\text{x}}}]\!](\overline{m})$ is equal to:

- $(b_{1\sqcup}, \emptyset)$ if $\overline{m} = (b_{1\sqcup}, ns)$ and $\overline{\text{x}}$ is *null*;

- $(b_{i_\sqcup}, \emptyset)$ if $\overline{m} = (b_1 p_1 ... b_{i_\sqcup}, ns)$ and $\overline{\mathrm{x}}$ is *null*;
- $(s[b_k]^{\mathrm{s}}, \emptyset)$ if $\overline{m} = (b_1 p_1 ... b_{i_\sqcup}, ns)$, $\overline{\mathrm{x}}$ may be not *null* and $\exists k : k = min\{z \in [1,i) : \overline{\mathrm{x}}$ *appears in* $p_z\}$;
- $\top_{\mathbf{M}}$ otherwise.

Theorem 1 (Soundness of the abstract semantics). $\mathfrak{S}_{\mathbf{M}}$ *and* $\mathfrak{L}_{\mathbf{M}}$ *are sound over-approximations of* $\mathfrak{S}$ *and* $\mathfrak{L}$ *respectively. Formally,*

$$\gamma_{\mathbf{M}}(\mathfrak{S}_{\mathbf{M}}[\![stm]\!](\overline{m})) \supseteq \{\mathfrak{S}[\![stm]\!](c) : c \in \gamma_{\mathbf{M}}(\overline{m})\}$$
$$\gamma_{\mathbf{B}}(\mathfrak{L}_{\mathbf{M}}[\![stm]\!](\overline{m})) \supseteq \{\mathfrak{L}[\![stm]\!](c) : c \in \gamma_{\mathbf{M}}(\overline{m})\}$$

where $c = \mu(\mathtt{c})$ *denotes the concrete value of* $\mathtt{c}$.

Proof. We prove the soundness separately for each operator.

- See Appendix A.4 (Theorem 2) for the `strcat` proof of soundness.
- Consider the unary operator `strlen`, and let $\overline{m}$ be a split segmentation abstract predicate. We have to prove that $\gamma_{\mathbf{B}}(\mathfrak{L}_{\mathbf{M}}[\![\mathtt{strlen}]\!](\overline{m})) \supseteq \{\mathfrak{L}[\![\mathtt{strlen}]\!](c) : c \in \gamma_{\mathbf{M}}(\overline{m})\}$. The `strlen` of c, if c contains a well-formed string, returns an integer value n denoting the length of the sequence of characters before the first null one, $\top_{\mathbb{Z}}$ otherwise, by definition of $\mathfrak{L}$. Then n belongs to $\gamma_{\mathbf{B}}(\mathfrak{L}_{\mathbf{M}}[\![\mathtt{strlen}]\!](\overline{m}))$ because $\mathfrak{L}_{\mathbf{M}}[\![\mathtt{st\text{-}rlen}]\!](\overline{m})$ is equal to the least upper bound of all the segment bounds in $\overline{m}.s$ (included their inner values) in which a certainly or maybe *null* value is contained, if $\overline{m}$ highlights the presence of well-formed strings; otherwise, the abstract operator returns $\top_{\mathbf{B}}$, by definition of $\mathfrak{L}_{\mathbf{M}}$.
- Consider the unary operator $\mathtt{strchr}_{\overline{\mathrm{x}}}$, and let $\overline{m}$ be a split segmentation abstract predicate. We have to prove that $\gamma_{\mathbf{M}}(\mathfrak{S}_{\mathbf{M}}[\![\mathtt{strchr}_{\overline{\mathrm{x}}}]\!](\overline{m})) \supseteq \{\mathfrak{S}[\![\mathtt{strchr}_{\mathrm{x}}]\!](c) : c \in \gamma_{\mathbf{M}}(\overline{m})\}$. The $\mathtt{strchr}_{\mathrm{x}}$ of c returns, if x is present in c, a sub-array of c (i.e. *sub.c*) that goes from the first occurrence of x in c to the first occurrence of the null-terminating character included, null otherwise, by definition of $\mathfrak{S}$. Then *sub.c* belongs to $\gamma_{\mathbf{M}}(\mathfrak{S}_{\mathbf{M}}[\![\mathtt{strchr}_{\overline{\mathrm{x}}}]\!](\overline{m}))$ because $\mathfrak{S}_{\mathbf{M}}[\![\mathtt{strchr}_{\overline{\mathrm{x}}}]\!](\overline{m})$, if $\overline{m}$ highlights the presence of well-formed strings and $\overline{\mathrm{x}}$ appears in $\overline{m}.s$, is equal to a sub-segmentation of $\overline{m}.s$ that goes from the first appearance of $\overline{\mathrm{x}}$ in $\overline{m}.s$ to the end of $\overline{m}.s$, and $\alpha_{\mathbf{C}}(\mathtt{x}) = \overline{\mathrm{x}}$; otherwise, the abstract operator returns $\top_{\mathbf{M}}$, by definition of $\mathfrak{S}_{\mathbf{M}}$.

□

3 Program abstraction

Adapting M-String to the analysis of real-world C programs requires, first of all, a procedure that identifies string operations automatically. A subset of such operations then needs to be performed using abstract operations, carried out on a suitable abstract representation. The technique that captures this approach is known as abstract interpretation. A typical implementation is based on an interpreter in the programming language sense: it executes the program by directly performing the operations written down in the source code. However, instead of using concrete values and concrete operations on those values, part (or the entirety) of the computation is performed in an *abstract domain*, which over-approximates the semantics of the concrete program.

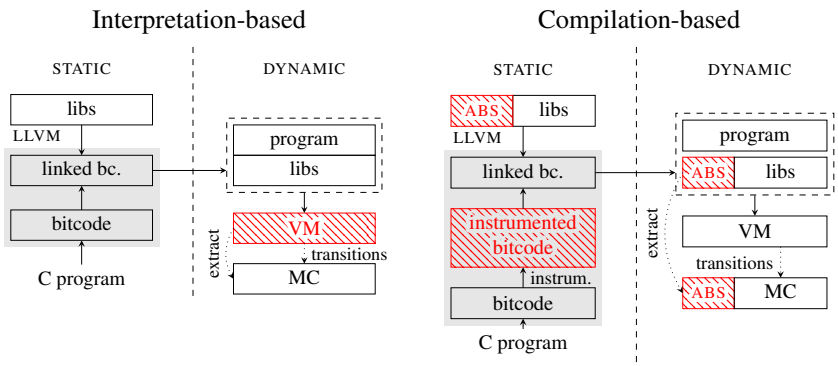

Fig. 1: The figure depicts a comparison of interpretation/compilation-based approaches. In interpretation-based approach, entire abstract interpretation is performed during runtime. A virtual machine (VM) interprets bitcode operations abstractly and maintain an abstract state. Consequently, it generates an abstract state-space for a model-checking algorithm (MC). On the other hand, compilation-based approach instruments abstract operations into the compiled program and provides their implementation as a library. A virtual machine then executes the instrumented program as regular bitcode.

Since in this paper, we focus on string abstraction, we would like to be able to perform the remainder of the program (i.e. the portions that do not work with strings) concretely. In fact, we only want to abstract some of the strings and string operations in the program, since the domain at hand is an approximation: in cases, where the program works with strings that exhibit minimal variation, e.g. string literals, using the M-String representation would not offer any benefit, and could actually hurt performance or introduce spurious counterexamples.

These considerations lead us to conclude that it would be beneficial to re-use, or rather re-purpose, existing tools which work with explicit programs to implement abstract interpretation in a modular fashion. A design in this style (compilation-based abtract interpretation) was proposed and implemented in [23].

However, as presented, the approach was limited to abstracting scalar values. In this paper, we extend this approach to work with strings and other domains that represent more complex objects.

3.1 Compilation-based approach

To perform abstraction, instead of (re-)interpreting instructions abstractly, we transform abstract instructions into equivalent explicit code, which implements the abstract computation. The transformation occurs before model checking (or other dynamical analysis), during the compilation process.

The transformed program can be further analyzed or processed without special knowledge of the abstract domains in use, because those are now encoded directly in the program. Comparison of this compilation-based approach and the approach of more

traditional abstract interpreters (an interpretation-based approach) is shown in Figure 1. In compilation-based approach, we consider two levels of abstraction:

1. *static*, concerning the syntax and the type system,
2. *dynamic*, or semantic, concerning execution and values.

LART performs syntactic (*static*) abstraction on LLVM bitcode [22]. The goal of syntactic abstraction is to replace some of the LLVM instructions in the program with their abstract counterparts. We illustrate syntactic abstraction in Figure 2.

3.2 Syntactic abstraction

During syntactic abstraction, LART performs a data flow analysis, starting from annotated abstract values (`abstract`) as the roots. The result of this analysis is the set of all operations that may come into contact with an abstract value. These are then substituted by their abstract counterparts (`a_strcat`, `a_strlen`). An abstract instruction takes abstract values as its inputs and produces an abstract value as its result. The specific meaning of those abstract instructions and abstract values then defines the semantic abstraction.

To formulate syntactic abstraction unambiguously, we take advantage of the static type system of LLVM. By assigning types to program variables, we can maintain a precise boundary between concrete and abstract values in our program.

We recognize a set of *concrete scalar types* S. We give a map Γ that inductively defines finite (non-recursive) algebraic types over the set of given scalars. To be specific, the set of all types $\Gamma(T)$ derived from a set of scalars T is defined as follows:

1. $T \subseteq \Gamma(T)$, meaning each scalar type is included in $\Gamma(T)$,
2. if $t_1, \ldots, t_n \in \Gamma(T)$ then also the *product type* is in $\Gamma(T)$: $(t_1, \ldots, t_n) \in \Gamma(T), n \in \mathbb{N}$,
3. if $t_1, \ldots, t_n \in \Gamma(T)$ then also *disjoint union* is in $\Gamma(T)$: $t_1 \mid t_2 \mid \cdots \mid t_n \in \Gamma(T), n \in \mathbb{N}$,
4. if $t \in \Gamma(T)$ then $t^* \in \Gamma(T)$, where t^* denotes pointer type.

In syntactic abstraction, we extend the concrete set of types by abstract types. From these, we generate admissible types using Γ. Depending on the level of abstraction, we define a different set of basic abstract types. In the case of scalar abstraction, a set of basic abstract types contains abstract scalar types A. Correspondence between abstract and concrete scalars is given by a bijective map $\Lambda : S \to A$. Finally, each value, which exists in the abstracted program, has an assigned type of $\Gamma(S \cup A)$. In particular, this means that the abstraction works with *mixed types* – products and unions with both concrete and abstract fields. Likewise, it is possible to form pointers to both abstract values and to mixed aggregates.

Concrete program:

```
a:str ← abstract()
b:str ← string()
c:str ← strcat(a,b)
l:int ← strlen(c)
```

Transformed program:

```
a:a_str ← a_string()
b:str ← string()
c:a_str ← a_strcat(a,b)
l:a_int ← a_strlen(c)
```

Fig. 2: Syntactic abstraction.

3.3 Aggregate domains

Scalars in a program are simple values which cannot be further decomposed into meaningful constituent parts. A typical example would be an integer, or a pointer. However,

programs typically also work with more complex data, that we can think of as compositions – aggregates – of multiple scalar values. Depending on the nature of such aggregates, we can classify them as arrays, which contain a variable number of items which all belong to a single type, records (structures), which contain a fixed number of items in a fixed layout, but each of these can be of a different type. The items in such aggregates can be (and often are) scalars, but more complicated aggregates are also possible: arrays of records, records which in turn contain other records, and so on.

In contrast to scalar domains, which deal with scalar values, an *aggregate domain* represents composite data, in the spirit of the above definition. An *abstract* aggregate domain approximates (concrete) aggregate values by keeping track of certain properties of the aggregate, for instance the length of an array, or a set of scalars that appear in the array. In the case of M-String, the information it tracks is a segmentation, where segments are represented using their bounds and a single value abstracting their content.

Aggregate domains could be equipped with quite arbitrary operations, though there are two that stand out, because they are in some sense universal, and those are byte-wise access and modification (update) of the content of the aggregate. The universality of those operations stems from the fact that in a low-level representation of a program, all operations with aggregate values take this form. In LLVM, it is possible (though not guaranteed), that access to the aggregate is encoded at a slightly higher level: as extraction and modification of entire scalars (as opposed to individual bytes). For M-String, though, this distinction is not important: the scalars stored in C strings are individual bytes. It should be also noted that the *access* and *update* form the interface between scalars and aggregates (even in the case of byte-oriented access, since bytes are also scalars). Therefore, the types of those two operations contain a single aggregate and (at least) a single scalar domain. Some (or all) of those domains may be abstract domains.

Syntactic abstraction has to handle aggregate domains differently from scalar domains. In LLVM, aggregate values are usually represented using pointers of a specific (aggregate) type. For this reason, aggregate abstraction starts from the types that represent its objects. In the case of arrays, those are concrete pointers into those arrays: let us call them P^*, where $P \subseteq \Gamma(S)$. We use the set of abstract pointers A^* to represent the types of abstract values in an aggregate domain. Thus the set of admissible types in the abstract program is generated by $\Gamma(S \cup A^*)$. Like in scalar domains, we define a natural correspondence between pointers to concrete values P^* as a bijective map $\Lambda : P^* \to A^*$.

Please note that pointers in general contain two pieces of information: they determine the *object* and an *offset* into that object. In explicit programs, this distinction is not very important, since those two parts are represented uniformly and often cannot be distinguished at all. The distinction, however, becomes important when we deal with abstract aggregate values. In this case, the *object* portion of the pointer is concrete, since it determines a single specific abstract object. However, the *offset* may or may not be concrete – depending on the specific abstract aggregate domain, it may be more advantageous to represent the offset abstractly. In either case, however, all memory access through such a pointer needs to be treated as an abstract *access* or *update* operation.

In LLVM, there are two basic memory access operations – *load* and *store*, which correspond to the *access* and *update* operations. Rather importantly, memory access is always explicit – memory is never directly used in a computation. We use this fact in the

design of aggregate abstraction, where we can assume that access to the content of an aggregate will always go through a pointer associated with the abstract object.

3.4 Semantic abstraction

Where syntactic abstraction was concerned with the syntax of operations, their types and the types of values and variables, semantic abstraction is concerned with the runtime values that appear during the computation performed by a program. While syntactic abstraction introduced the maps Λ and Λ^{-1} to transfer between concrete and abstract *types*, semantic abstraction introduces *lift* and *lower*: operations (instructions) which convert between concrete and abstract *values*. They represent a realization of the abstraction (α) and concretization (γ) functions.

While *lift* and *lower* form a boundary between concrete and abstract scalar computation, the *access* and *update* operations of an aggregate domain form a boundary between scalar and aggregate domains. We kindly refer to [23], where a reader may find how LART transforms an abstract program into an executable form.

3.5 Abstract operations

After syntactic abstraction, the program temporarily contains abstract instructions. Abstract instructions take abstract values as operands and give back abstract values as their results. However, after transformation, we require that the resulting program is semantically valid LLVM bitcode. Hence, it is crucial that each abstract instruction can be realized as a suitable sequence of concrete instructions. This makes it possible to obtain an abstract program that does not actually contain any abstract instructions and execute it using standard (concrete, explicit) methods.

In detail, syntactic abstraction replaces concrete instructions with their abstract counterparts: an instruction with type $(t_1,\ldots,t_n) \to t_r$ is substituted by an abstract instruction of type $(\Lambda(t_1),\ldots,\Lambda(t_n)) \to \Lambda(t_r)$. Moreover, *lift* and *lower* are inserted as needed. The implementation is free to decide which instructions to abstract and where to insert value lifting and lowering, so long as it obeys type constraints.

Additionaly, in string abstraction, we also want to abstract function calls such as `strcat`, `strcpy` etc. From the perspective of abstraction, we treat these functions as single operations that take abstract values and produce results. Therefore, we can process them in the same way as instructions. For example, by transforming `strcat` of type $(str, str) \to str$ we obtain $strcat_a$ of type $(\Lambda(str), \Lambda(str)) \to \Lambda(str)$. Afterwards, all abstract operations are realized using concrete subroutines [23].

We could have also transformed standard library functions (`strcat`, `strcmp`, etc.) instruction by instruction using only abstract access and update of a content, but in this way we would lose a certain degree of precision in the abstraction, the exact amount depending on the operation.

4 Instantiating M-String

M-String, as a content domain, enables a parametrization of string abstraction. To be specific, it supports the parametrization of string segmentation representation in which

we can substitute different domains of bounds and characters. As a representation of string values, we can use a scalar domain equipped with the correct operations, and the same holds for bounds of segments as described in section 2.

An implementation of a particular M-String instance can be automatically derived from a parametric description, given well-defined abstract domains **C** for characters and **B** to represent segment bounds. M-String also requires that both **C** and **B** support certain operations that appear in the generic implementation of the abstract operations. These are mainly basic arithmetic and relational operators. For further details of the implementation, see the appendix of this paper.

4.1 Symbolic scalar values

In program verification, it is common practice to represent certain values symbolically (for instance inputs from the environment). This type of representation enables a verification procedure to consider all the possible values with a reasonably small overhead. computation is implemented using abstraction of the same type as described here: computations on scalar values are lifted into the term domain, which simply keeps track of values using terms (expressions) in form of abstract syntax trees. Those trees contain atoms (unconstrained values) and operators of the bitvector logic. The term domain additionally keeps track of any constraints derived from the control flow of the program (a *path condition*). A more detailed description is presented in [23].

Paired with a constraint solver for the requisite theory,[4] the term domain coincides with symbolic computation. The solver makes it possible to detect computations that have reached the bottom of the term domain (those are the infeasible paths through the program) and also to check for equality or subsumption of program states. With those provisions, the bitvector theory is completely precise (i.e. it is not an approximation, but rather models the program state faithfully).

4.2 Concrete characters, symbolic bounds

For evaluation purposes, we have instantiated the M-String domain by setting **C**, the domain of the individual characters, to be the concrete domain (i.e. characters are represented by themselves) and **B**, the domain of segment bounds, to be symbolic 64b integers. The main motivation for this instantiation is a balance between simplicity on one hand (both the domains we used for parameters were already available in the tools we used) and the ability to describe strings with undetermined length and structure.

At the implementation level (as explained in more detail in the following section), the domain continues to be parametric: the specific domains we picked could be easily swapped for other domains (an immediate candidate would be using both symbolic characters and symbolic bounds). Compared to the theoretical description of M-String, the implementation uses a slightly simplified representation using a pair of arrays (cf. Figure 3), where the specific type of characters and bounds is given by the parameter domains **C** and **B** respectively.

[4] For scalars in C programs, we use the bitvector theory.

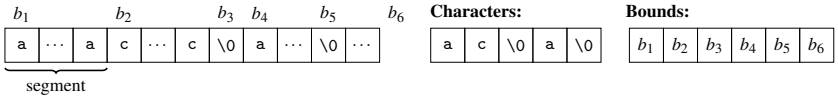

Fig. 3: M-String value with symbolic bounds, where string of interest is from b_1 to b_3.

	Word						Sequence						Alternation					
		verification(s)						verification(s)						verification(s)				
	states	8	64	1024	4096	LART(s)	states	8	64	1024	4096	LART(s)	states	8	64	1024	4096	LART(s)
`strcmp`	163	12.8	14.7	17.1	27.2	0.85	12	0.55	0.67	1.15	1.79	0.65	744	63.4	110	129	141	0.77
`strcpy`	36	1.67	1.63	2.18	2.48	0.51	9	0.36	0.27	0.51	0.83	0.88	74	3.62	4.9	5.3	4.36	0.46
`strcat`	477	32.2	33.4	31.9	33.4	0.92	25	2.28	2.5	2.79	3.19	0.93	2406	208	218	220	205	0.95
`strchr`	24	0.28	0.35	0.53	1.14	0.88	6	0.03	0.08	0.13	0.26	0.56	45	0.54	0.54	0.92	1.89	0.83
`strlen`	26	0.45	0.46	0.69	1.31	0.86	6	0.09	0.11	0.17	0.33	0.86	53	1.01	1.21	1.94	2.33	0.82

Table 1: Benchmarks of abstract operations were evaluated on three types of M-Strings (*Word*, *Sequence*, and *Alternation*) – see section 5 for description. The table depicts the number of states in the state space of the verified program, verification time in seconds for the different length of inputs and an average time of a transformation (LART).

M-String, when instantiated like this, is particularly suitable for representing strings with runs of a single character of variable length, i.e. the strings of the form $a^k b^l c^m ...$ where relationships between $k, l, m, ...$ can be specified using standard arithmetic and relational operators and each of a, b, c is a specific letter. This in turn allows M-String to be used for checking program behaviour on broad classes of input strings described this way. A more detailed account of this approach can be found in Section 5.

4.3 Implementation

We have implemented the abstract semantics of operations in the M-String domain as a C++ library, in a form that allows programs to be automatically lifted into this domain by LART and later model-checked with DIVINE. An abstract domain definition in LART consists of a C++ class that describes both the representation (in terms of data) and the operations (in terms of code) of the abstract domain.

The abstract domain is equipped with a set of essential operations, which appear in all programs that work with strings: these are *lift*, *update* and *access*. All other operations which involve strings can be, in principle, derived automatically using the same procedure that is applied to user programs. However, abstracting only *access* and *update* causes either a loss of precision or a blowup in complexity. For this reason, we also include hand-crafted implementations of the following abstract operations: `strcmp`, `strcpy`, `strcat`, `strchr`, and `strlen`. These are all based on the abstract semantics of the respective operations as described in Section 2 and in the Appendix A.4.

A more complete description of the implementation of LART and DIVINE, their source code, and the Appendices A.1–A.4 which describe the technical details of the MString domain can be found online.[5]

5 Experimental evaluation

For evaluation purposes, we have picked three scenarios. In first of those, we show that the provided implementation of basic string functions is more efficient than lifting them

[5] https://divine.fi.muni.cz/2019/mstring

	Word						*Sequence*						*Alternation*					
length	4		8		16		4		8		16		4		8		16	
`strcmp`	18.5s	90	597s	1228	2410s	11416	3.82s	14	12s	32	32.2s	68	70.7s	348	T	–	T	–
`strcpy`	8.4s	45	99s	438	775s	4410	5.7s	14	17.5s	24	71.8s	44	11.6s	80	168s	928	3230s	19234
`strcat`	75.5s	303	T	–	T	–	23.7s	35	117s	149	769s	737	249s	1085	T	–	T	–
`strchr`	12.4s	39	166s	245	934s	1265	4.34s	8	16.5s	12	158s	20	13.4s	57	316s	815	T	–
`strlen`	0.5s	27	7.9s	169	811s	1365	0.27s	8	0.6s	14	1.7s	20	1.8s	48	69s	357	3250s	5307

Table 2: The table depicts the verification results of functions from `pdclib`. For each type of input M-String of a given length, we present duration of verification and the size of the state space. T denotes a timeout of the verification.

automatically based on the *access* and *update* operations. In the second scenario, we analyse various implementations of the same string functions by lifting them automatically and checking that their outputs match the ones we expect based on the concrete semantics of those operations – in this case, the inputs are provided in the form of specific abstract (M-String) values. In the last scenario, we have picked a few real-world programs to demonstrate that M-String can be successfully used in analysis of moderately complex C code. To this end, we have chosen two context-free grammars and used them to generate C parsers using the `bison` and `flex` tools, again providing abstract strings as inputs to the generated parsers. All experiments were performed with an identical set of resource constraints: 1 hour of CPU time, 80 GB of RAM and 4 CPU cores.[6]

Abstract operations: The first set of benchmarks covers resource usage measurements of M-String operations. Results are presented in Table 1. We run each operation separately on three different M-String inputs with a single parameter, *length*:

- *Word* is a string of the form $a^i b^j c^k$, $i+j+k \leq length$,
- *Sequence* has the form a^{length}, and
- *Alternation* is $a^i b^j a^k b^l$, $i+j+k+l \leq length$.

We have measured how much time we spend in the abstract operations which are part of the M-String domain and compare them to the same programs, but with the functions abstracted automatically, using only the M-String definitions of *access* and *update*.

One of the results is that the size of the state space does not depend on the length of the string when using the operations from M-String. This is because the number of segments does not change and the operations perform the same amount of work.In comparison, analysis of automatically lifted implementations of the same functions[7] does not terminate in a 1-hour time limit for strings of length 64 and more. This is caused by the fact that the concrete implementations need to iterate over each character individually, while the M-String implementation directly works with segments.

C standard library: The second scenario deals with correctness of various concrete implementations of the same set of standard library functions. Namely, we used 3 sources: `pdclib`, `musl-libc` and μCLibc. The results are very similar, hence we only present results for `pdclib` – data for the remaining 2 are part of the supplementary material.

[6] The processor used to run the benchmarks was Intel Xeon E5-2630 clocked at 2.60GHz. To make reproduction of the benchmarks easier, we provide instructions and scripts in the online supplementary material.

[7] The implementations were taken from `pdclib`, a public-domain `libc` implementation.

Numeric Expression Grammar

length	10		20		25		35	
Add.	40.2s	416	319s	3548	622s	13k	T	–
Ones	5.5s	62	8.1s	196	29.7s	402	189s	2186
Alter.	708s	105	582s	11k	T	–	T	–

BP Grammar

length	10		50		100		1000	
Value	6.58s	38	44.4s	238	90.4s	488	1100s	4988
Loop	1.53s	23	3.28s	23	4.88s	23	33.3s	23
Wrong	7.34s	82	27.9	442	67.7s	892	311s	8992

Table 3: Evaluation on parsers of mathematical expressions (ME) and simple programs (BP). Inputs for ME were of 3 forms: *Addition* is a string with two numbers with + between them, *Ones* is a sequence of ones, and *Alternation* represent a number with multiple digits. Inputs for BP were of the form: *Value* constructs a constant, while *Loop* is a program with a single bounded loop and *Wrong* is a program with a syntax error.

In these benchmarks, we compare the results of the abstract implementation with the result of the automatically abstracted (originally concrete) implementation of each function and check that they give identical results.

Results show that analysis of strings with alternating characters is more expensive. This is because a segment might disappear and two segments are merged into one: the SMT queries arising from those events are hard to solve, because of the large number of possible overlaps in the segment bounds.

The library implementations access and update the string one character at a time, resulting in large SMT formulas – this causes the blowup in analysis time and hence timeouts with longer strings.

Bison grammar: In the last scenario, we analyse two parsers generated by `bison`. First is a parser for numerical expressions which consist of binary operators and numbers (see Table 3). The second example is a parser for a simple programming language.

Like with the previous scenarios, inputs which contain long sequences of the same character perform the best, especially when contrasted with a similar task performed on an input with alternating digits.

6 Conclusion

We have presented a segmentation-based abstract domain for approximating C strings. The main novelty of the domain lies in its focus on string buffers, which consist of two parts: the string of interest itself, and a tail of allocated and possibly initialized but unused memory. This paradigm allows for precise modeling of string functions from the standard C library, including their often fragile handling of terminating zeroes and buffer bounds. In principle, this allows the M-String domain to identify string manipulation errors with security consequences, such as buffer overflows.

In addition to presenting the domain theoretically, we have implemented the abstract semantics in executable form (as C++ code) and combined them with a tool that automatically lifts string-manipulating code in existing C programs to the M-String domain. Since M-String is a parametric domain – the domains for both segment content and segment bounds can be freely chosen – we have instantiated M-String (for evaluation purposes) with concrete characters and with symbolic (bitvector) bounds.

References

1. Polyspace, MathWorks, 2001.
2. Static Code Analysis, OWASP, 2017.
3. Interactive: The Top Programming Languages 2018, IEEE Spectrum Magazine, 2018.
4. Roberto Amadini, Graeme Gange, François Gauthier, Alexander Jordan, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Reference Abstract Domains and Applications to String Analysis. *Fundam. Inform.*, 158(4):297–326, 2018.
5. Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 41–57, 2017.
6. Zuzana Baranová, Jiri Barnat, Katarína Kejstová, Tadeás Kucera, Henrich Lauko, Jan Mrázek, Petr Rockai, and Vladimír Still. Model Checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, pages 201–207, 2017.
7. Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String Analysis for Software Verification and Security*. Springer, 2017.
8. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pages 1–18, 2003.
9. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
10. Agostino Cortesi and Martina Olliaro. M-String Segmentation: a Refined Abstract Domain for String Analysis in C Programs. In *Theoretical Aspects of Software Engineering - 12th International symposium, TASE 2018, Guangzhou, China, August 29-31, 2018, Proceedings*, 2018.
11. Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. A Suite of Abstract Domains for Static Analysis of String Values. *Softw., Pract. Exper.*, 45(2):245–287, 2015.
12. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.
13. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *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, 2011.
14. Nurit Dor, Michael Rodeh, and Shmuel Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 155–167, 2003.
15. Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
16. David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.
17. Gerard J. Holzmann. Static Source Code Checking for User-Defined Properties. Integrated Design and Process Technology, IDPT-2002, Society for Design and Process Science, Pasadena, CA, USA, 2002.

18. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009.
19. Richard W. M. Jones and Paul H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*, pages 13–26, 1997.
20. Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 121–132, 2014.
21. Se-Won Kim, Wooyoung Chin, Jimin Park, Jeongmin Kim, and Sukyoung Ryu. Inferring Grammatical Summaries of String Values. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 372–391, 2014.
22. Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", booktitle = "International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, Mar 2004.
23. Henrich Lauko, Petr Rockai, and Jiri Barnat. Symbolic Computation via Program Transformation. In *Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*, pages 313–332, 2018.
24. Magnus Madsen and Esben Andreasen. String Analysis for Dynamic Field Access. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 197–217, 2014.
25. Aleph One. Smashing The Stack For Fun And Profit. Phrack Magazine, 1996.
26. Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 25–36, 2016.
27. Hossain Shahriar and Mohammad Zulkernine. Classification of Static Analysis-Based Buffer Overflow Detectors. In *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume*, pages 94–101, 2010.
28. Fausto Spoto. The Julia Static Analyzer for Java. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 39–57, 2016.
29. 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*, 2000.
30. Yichen Xie, Andy Chou, and Dawson R. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, pages 327–336, 2003.
31. Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 27–38, 2008.