Proving Termination and Memory Safety for Programs with Pointer Arithmetic

Thomas Ströder¹, Jürgen Giesl¹, Marc Brockschmidt², Florian Frohn¹, Carsten Fuhs³, Jera Hensel¹, and Peter Schneider-Kamp⁴

¹ LuFG Informatik 2, RWTH Aachen University, Germany
² Microsoft Research Cambridge, UK
³ Dept. of Computer Science, University College London, UK
⁴ IMADA, University of Southern Denmark, Denmark

Abstract. Proving termination automatically for programs with explicit pointer arithmetic is still an open problem. To close this gap, we introduce a novel abstract domain that can track allocated memory in detail. We use it to automatically construct a symbolic execution graph that represents all possible runs of the program and that can be used to prove memory safety. This graph is then transformed into an integer transition system, whose termination can be proved by standard techniques. We implemented this approach in the automated termination prover AProVE and demonstrate its capability of analyzing C programs with pointer arithmetic that existing tools cannot handle.

1 Introduction

Consider the following standard C implementation of strlen [23, 30], computing the length of the string at pointer str. In C, strings are usually represented as a pointer str to the heap, where all following memory cells up to the first one that contains the value 0 are allocated memory and form the value of the string.

```c
int strlen(char* str) { char* s = str; while(*s) s++; return s-str; }
```

To analyze algorithms on such data, one has to handle the interplay between addresses and the values they point to. In C, a violation of memory safety (e.g., dereferencing NULL, accessing an array outside its bounds, etc.) leads to undefined behavior, which may also include non-termination. Thus, to prove termination of C programs with low-level memory access, one must also ensure memory safety. The strlen algorithm is memory safe and terminates because there is some address end ≥ str (an integer property of end and str) such that *end is 0 (a pointer property of end) and all addresses str ≤ s ≤ end are allocated. Other typical programs with pointer arithmetic operate on arrays (which are just sequences of memory cells in C). In this paper, we present a novel approach to prove memory safety and termination of algorithms on integers and pointers automatically. To avoid handling the intricacies of C, we analyze programs in the platform-independent intermediate representation (IR) of the LLVM compilation framework [17]. Our approach works in three steps: First, a symbolic execution graph is created

* Supported by DFG grant GI 274/6-1 and Research Training Group 1298 (AlgoSyn).
that represents an over-approximation of all possible program runs. We present our abstract domain based on separation logic \[22\] and the automated construction of such graphs in Sect. 2. In this step, we handle all issues related to memory, and in particular prove memory safety of our input program. In Sect. 3, we describe the second step of our approach, in which we generate an integer transition system (ITS) from the symbolic execution graph, encoding the essential information needed to show termination. In the last step, existing techniques for integer programs are used to prove termination of the resulting ITS. In Sect. 4, we compare our approach with related work and show that our implementation in the termination prover AProVE proves memory safety and termination of typical pointer algorithms that could not be handled by other tools before.

2 From LLVM to Symbolic Execution Graphs

In Sect. 2.1, we introduce concrete LLVM states and abstract states that represent sets of concrete states, cf. [9]. Based on this, Sect. 2.2 shows how to construct symbolic execution graphs automatically. Sect. 2.3 presents our algorithm to generalize states, needed to always obtain finite symbolic execution graphs.

To simplify the presentation, we restrict ourselves to a single LLVM function without function calls and to types of the form \(i_n\) (for \(n\)-bit integers), \(i_n^*\) (for pointers to values of type \(i_n\)), \(i_n^{**}\), \(i_n^{***}\), etc. Like many other approaches to termination analysis, we disregard integer overflows and assume that variables are only instantiated with signed integers appropriate for their type. Moreover, we assume a 1 byte data alignment (i.e., values may be stored at any address).

2.1 Abstract Domain

Consider the \texttt{strlen} function from Sect. 1. In the corresponding LLVM code,\(^5\) \texttt{str} has the type \texttt{i8*}, since it is a pointer to the string’s first character (of type \texttt{i8}). The program is split into the basic blocks \texttt{entry}, \texttt{loop}, and \texttt{done}. We will explain this LLVM code in detail when constructing the symbolic execution graph in Sect. 2.2.

Concrete LLVM states consist of the program counter, the values of local variables, and the state of the memory. The program counter is a 3-tuple \((b_{\text{prev}}, b, i)\), where \(b\) is the name of the current basic block, \(b_{\text{prev}}\) is the previously executed

\begin{verbatim}
define i32 @strlen(i8* str) {
  entry: 0: c0 = load i8* str
         1: c0zero = icmp eq i8 c0, 0
         2: br i1 c0zero, label done, label loop
  loop: 0: olds = phi i8* [str,entry],[s,loop]
         1: s = getelementptr i8* olds, i32 1
         2: c = load i8* s
         3: czero = icmp eq i8 c, 0
         4: br i1 czero, label done, label loop
  done: 0: sfin = phi i8* [str,entry],[s,loop]
         1: sfinint = ptrtoint i8* sfin to i32
         2: strint = ptrtoint i8* str to i32
         3: size = sub i32 sfinint, strint
         4: ret i32 size }
\end{verbatim}

\(^5\) This LLVM program corresponds to the code obtained from \texttt{strlen} with the Clang compiler [8]. To ease readability, we wrote variables without “\%” in front (i.e., we wrote “\texttt{str}” instead of “\texttt{\%str}” as in proper LLVM) and added line numbers.
block,\(^6\) and \(i\) is the index of the next instruction. So if \(\text{Blks}\) is the set of all basic blocks, then the set of code positions is \(\text{Pos} = (\text{Blks} \cup \{\varepsilon\}) \times \text{Blks} \times \mathbb{N}\). We represent assignments to the local program variables \(\mathcal{V}_p\) (e.g., \(\mathcal{V}_p = \{\text{str}, \text{c0}, \ldots\}\)) as functions \(s : \mathcal{V}_p \to \mathbb{Z}\). The state of the memory is represented by a partial function \(m : \mathbb{N}_{>0} \to \mathbb{Z}\) with finite domain that maps addresses to integer values. So a concrete LLVM state is a 3-tuple \((p, s, m)\) for all addresses in \(m\)’s domain are allocated. Let \(\to_{\text{LLVM}}\) be LLVM’s evaluation relation on concrete states, i.e., \((p, s, m) \to_{\text{LLVM}} (p', s', m')\) holds iff \((p, s, m)\) evaluates to \((p', s', m')\) by executing one LLVM instruction. Similarly, \((p, s, m) \to_{\text{LLVM}} \text{ERR}\) means that the instruction at position \(p\) accesses an address where \(m\) is undefined. An LLVM program is memory safe for \((p, s, m)\) iff there is no evaluation \((p, s, m) \to_{\text{LLVM}}^+ \text{ERR}\), where \(\to_{\text{LLVM}}^+\) is the transitive closure of \(\to_{\text{LLVM}}\).

To formalize abstract states that stand for sets of concrete states, we use a fragment of separation logic [22]. Here, an infinite set of symbolic variables \(\mathcal{V}_{\text{sym}}\) with \(\mathcal{V}_{\text{sym}} \cap \mathcal{V}_p = \emptyset\) can be used in place of concrete integers. We represent abstract states as tuples \((p, \mathcal{V}_p, \mathcal{V}_{\text{sym}}, \mathcal{P}_0)\). Again, \(p \in \text{Pos}\) is the program counter. The function \(\mathcal{L} : \mathcal{V}_p \to \mathcal{V}_{\text{sym}}\) maps every local variable to a symbolic variable. To ease the generalization of states in Sect. 2.3, we require injectivity of \(\mathcal{L}\). The knowledge base \(\mathcal{KB} \subseteq \text{QF}_{\text{IA}}(\mathcal{V}_{\text{sym}})\) is a set of pure quantifier-free first-order formulas that express integer arithmetic properties of \(\mathcal{V}_{\text{sym}}\).

The allocation list \(\text{AL}\) contains expressions of the form \(\text{alloc}(v_1, v_2)\) for \(v_1, v_2 \in \mathcal{V}_{\text{sym}}\), which indicate that \(v_1 \leq v_2\) and that all addresses between \(v_1\) and \(v_2\) are allocated. Finally, \(\mathcal{P}_0\) is a set of “points-to” atoms \(v_1 \leftarrow_{\text{ty}} v_2\) where \(v_1, v_2 \in \mathcal{V}_{\text{sym}}\) and \(\text{ty}\) is an LLVM type. This means that the value \(v_2\) of type \(\text{ty}\) is stored at the address \(v_1\). Let \(\text{size}(\text{ty})\) be the number of bytes required for values of type \(\text{ty}\) (e.g., \(\text{size}(18) = 1\) and \(\text{size}(132) = 4\)). As each memory cell stores one byte, \(v_1 \leftarrow_{132} v_2\) means that \(v_2\) is stored in the four cells at the addresses \(v_1, \ldots, v_1 + 3\).

**Definition 1 (Abstract States).** Abstract states have the form \((p, \mathcal{L}, \mathcal{V}_p, \mathcal{V}_{\text{sym}}, \mathcal{P}_0)\) where \(p \in \text{Pos}\), \(\mathcal{L} : \mathcal{V}_p \to \mathcal{V}_{\text{sym}}\) is injective, \(\mathcal{KB} \subseteq \text{QF}_{\text{IA}}(\mathcal{V}_{\text{sym}})\), \(\mathcal{AL} \subseteq \{\text{alloc}(v_1, v_2) \mid v_1, v_2 \in \mathcal{V}_{\text{sym}}\}\), and \(\mathcal{P}_0 \subseteq \{(v_1 \leftarrow_{\text{ty}} v_2) \mid v_1, v_2 \in \mathcal{V}_{\text{sym}}, \text{ty is an LLVM type}\}\). Additionally, there is a state \(\text{ERR}\) for violations of memory safety.

We often identify \(\mathcal{L}\) with the set of equations \(\{x = \mathcal{L}(x) \mid x \in \mathcal{V}_p\}\) and extend \(\mathcal{L}\) to a function from \(\mathcal{V}_p \cup \mathcal{Z}\) to \(\mathcal{V}_{\text{sym}} \cup \mathcal{Z}\) by defining \(\mathcal{L}(z) = z\) for all \(z \in \mathcal{Z}\).

As an example, consider the following abstract state for the `strlen` program:

\[
( (\varepsilon, \text{entry}, 0), \{\text{str} = u_{\text{str}}, \ldots, \text{size} = u_{\text{size}}\}, \{z = 0\}, \{\text{alloc}(u_{\text{str}}, v_{\text{end}})\}, \{v_{\text{end}} \leftarrow_{18} z\} ).
\]

It represents states at the beginning of the `entry` block, where \(\mathcal{L}(x) = u_x\) for all \(x \in \mathcal{V}_p\), the memory cells between \(\mathcal{L}(\text{str}) = u_{\text{str}}\) and \(v_{\text{end}}\) are allocated, and the value at the address \(v_{\text{end}}\) is \(z\) (where the knowledge base implies \(z = 0\)).

To define the semantics of abstract states \(a\), we introduce the formulas \(\langle a \rangle_{\text{SL}}\) and \(\langle a \rangle_{F0}\). The separation logic formula \(\langle a \rangle_{\text{SL}}\) defines which concrete states are...
represented by \( a \). The first-order formula \( \langle a \rangle_{FO} \) is used to construct symbolic execution graphs, allowing us to use standard SMT solving for all reasoning in our approach. Moreover, we also use \( \langle a \rangle_{FO} \) for the subsequent generation of integer transition systems from the symbolic execution graphs. In addition to \( \langle KB, \langle a \rangle_{FO} \rangle \) states that the expressions \( alloc(v_1, v_2) \in AL \) represent disjoint intervals and that two addresses must be different if they point to different values in \( PT \).

In \( \langle a \rangle_{SL} \), we combine the elements of \( AL \) with the separating conjunction “\( \land \)” to ensure that different allocated memory blocks are disjoint. Here, as usual \( \varphi_1 \land \varphi_2 \) means that \( \varphi_1 \) and \( \varphi_2 \) hold for disjoint parts of the memory. In contrast, the elements of \( PT \) are combined by the ordinary conjunction “\( \land \)”.

Moreover, we also use the two formulas resulting from \( \langle a \rangle_{SL} \) to ensure that different allocated memory blocks are disjoint. Here, as usual “\( \land \)” as both express different properties of memory addresses.

**Definition 2 (Representing States by Formulas).** For \( v_1, v_2 \in V_{sym} \), let \( \langle alloc(v_1, v_2) \rangle_{SL} = v_1 \leq v_2 \land (\forall x. \exists y. (v_1 \leq x \leq v_2) \Rightarrow (x \mapsto y)) \). Due to the two’s complement representation, for any LLVM type \( ty \), we define \( \langle v_1 \mapsto_{ty} v_2 \rangle_{SL} = \langle v_1 \mapsto_{ty} v_3 \rangle_{SL} \land (v_2 \geq 0 \Rightarrow v_3 = v_2) \land (v_2 < 0 \Rightarrow v_3 = v_2 + 2^{8 \cdot size(ty)}) \),

where \( v_3 \in V_{sym} \) is fresh. Here, \( \langle v_1 \mapsto_{ty} v_3 \rangle_{SL} = true \) and \( \langle v_1 \mapsto_{ty} v_3 \rangle_{SL} = v_1 \mapsto (v_3 \mod 256) \land (\langle v_1 + 1 \rangle \mapsto_{ty} (v_3 \div 256)) \). Then \( a = (p, LV, KB, AL, PT) \) is represented by \( \langle a \rangle_{SL} = LV \land KB \land (\forall \varphi \in AL (\varphi)_{SL} \land (\forall \varphi \in PT (\varphi)_{SL}) \).

Moreover, the following first-order information on \( V_{sym} \) is deduced from an abstract state \( a = (p, LV, KB, AL, PT) \). Let \( \langle a \rangle_{FO} \) be the smallest set with

\[
\langle a \rangle_{FO} = KB \cup \{ v_1 \leq v_2 \mid alloc(v_1, v_2) \in AL \} \cup \\
\{ v_2 < w_1 \lor w_2 < v_1 \mid alloc(v_1, v_2), alloc(w_1, w_2) \in AL, (v_1, v_2) \neq (w_1, w_2) \} \cup \\
\{ v_1 \neq w_1 \mid (v_1 \mapsto_{ty} v_2), (w_1 \mapsto_{ty} w_2) \in PT \land (\langle a \rangle_{FO} \Rightarrow v_2 \neq w_2) \}.
\]

Let \( T(V_{sym}) \) be the set of all arithmetic terms containing only variables from \( V_{sym} \). Any function \( \sigma : V_{sym} \rightarrow T(V_{sym}) \) is called an instantiation. Thus, \( \sigma \) does not instantiate \( \forall \varphi \). Instantiations are extended to formulas in the usual way, i.e., \( \sigma(\varphi) \) instantiates every \( v \in V_{sym} \) that occurs free in \( \varphi \) by \( \sigma(v) \). An instantiation is called concret if \( \sigma(v) \in Z \) for all \( v \in V_{sym} \). Then an abstract state \( a \) at position \( p \) represents those concrete states \( (p, s, m) \) where \( (s, m) \) is a model of \( \sigma(a)_{SL} \) for a concrete instantiation \( \sigma \) of the symbolic variables. So for example, the abstract state \( (\xi) \) on the previous page represents all concrete states ((\epsilon, entry, 0), s, m) where \( m \) is a memory that stores a string at the address \( s(str) \).

---

7 We assume a little-endian data layout (where least significant bytes are stored in the lowest address). A corresponding representation could also be defined for big-endian layout. This layout information is necessary to decide which concrete states are represented by abstract states, but it is not used when constructing symbolic execution graphs (i.e., our remaining approach is independent of such layout information).

8 We identify sets of first-order formulas \( \{ \varphi_1, \ldots, \varphi_n \} \) with their conjunction \( \varphi_1 \land \ldots \land \varphi_n \).

9 The reason is that then there is an address end \( \geq s(str) \) such that \( m(end) = 0 \) and \( m \) is defined for all numbers between \( s(str) \) and end. Hence, \( (s, m) \models \sigma(a)_{SL} \) holds for an instantiation with \( \sigma(u_x) = s(x) \) for all \( x \in V_F \), \( \sigma(v_{end}) = end \), and \( \sigma(z) = 0. \)
We use replacing all $s,m$ from $V$ and quantifiers is as usual. In particular, we define semantics of arithmetic operations and relations and of first-order connectives $\psi$ are implicitly universally quantified, i.e., $m \models \forall \psi$ iff $m \models \forall \sigma_1, \ldots, \sigma_n. \psi$. The semantics of arithmetic operations and relations and of first-order connectives and quantifiers is as usual. In particular, we define $m \models \forall v. \psi$ iff $m \models \sigma(v)$ holds for all instantiations $\sigma$ where $\sigma(v) \in \mathbb{Z}$ and $\sigma(w) = w$ for all $w \in \mathcal{V}_{sym} \setminus \{v\}$.

We still have to define the semantics of $\rightarrow$ and $\ast$ for variable-free formulas. For $z_1, z_2 \in \mathbb{Z}$, let $m \models z_1 \rightarrow z_2$ hold iff $m(z_1) = z_2$.\footnote{We use $\rightarrow_\ast$ instead of $\rightarrow_\psi$ in separation logic, since $m \models z_1 \rightarrow z_2$ would imply that $m(z)$ is undefined for all $z \neq z_1$. This would be inconvenient in our formalization, since $PT$ usually only contains information about a part of the allocated memory.} The semantics of $\ast$ is defined as usual in separation logic: For two partial functions $m_1, m_2 : \mathbb{N}_{\geq 0} \rightarrow \mathbb{Z}$, we write $m_1 \downarrow m_2$ to indicate that the domains of $m_1$ and $m_2$ are disjoint and $m_1 \cdot m_2$ denotes the union of $m_1$ and $m_2$. Then $m \models \varphi_1 \ast \varphi_2$ iff there exist $m_1 \downarrow m_2$ where $m_1 \models \varphi_1$ and $m_2 \models \varphi_2$.

As usual, $\models \varphi$ means that $\varphi$ is a tautology, i.e., that $(s,m) \models \varphi$ holds for any $s : \mathcal{V}_P \rightarrow \mathbb{Z}$ and $m : \mathbb{N}_{\geq 0} \rightarrow \mathbb{Z}$. Clearly, $\models \langle a \rangle_{SL} \Rightarrow \langle a \rangle_{FO}$, i.e., $\langle a \rangle_{FO}$ contains first-order information that holds in every concrete state represented by $a$.

2.2 Constructing Symbolic Execution Graphs

We now show how to automatically generate a symbolic execution graph that over-approximates all possible executions of a given program. For this, we present symbolic execution rules for some of the most important LLVM instructions. Other instructions can be handled in a similar way, cf. [26]. Note that in contrast to other formalizations of LLVM’s operational semantics [31], our rules operate on abstract instead of concrete states to allow a symbolic execution of LLVM. In particular, we also have rules for refining and generalizing abstract states.

Our analysis starts with the set of initial states that one wants to analyze for termination, e.g., all states where str points to a string. So in our example, we start with the abstract state $(\dagger)$. Fig. 1 depicts the symbolic execution graph for strlen. Here, we omitted the component $AL = \{alloc(u_{str}, v_{end})\}$, which stays the same in all states in this example. We also abbreviated parts of $LV$, $KB$, $PT$ by “...”. Instead of $v_{end} \leftarrow 18 \quad z$ and $z = 0$, we directly wrote $v_{end} \leftarrow 0$, etc.

The function strlen starts with loading the character at address str to c0. Let $p : ins$ denote that $ins$ is the instruction at position $p$. Our first rule handles the case $p : \text{“x = load ty\* ad”}$, i.e., the value of type $ty$ at the address $ad$ is assigned to the variable $x$. In our rules, $a$ always denote the abstract state before the execution step (i.e., above the horizontal line of the rule). Moreover, we write $\langle a \rangle$ instead of $\langle a \rangle_{FO}$. As each memory cell stores one byte, in the load-rule we first have to check whether the addresses $ad, \ldots, ad + size(ty) - 1$ are allocated, i.e., if there is an $alloc(v_1, v_2) \in AL$ such that $\langle a \rangle \Rightarrow (v_1 \leq LV(ad) \land \langle a \rangle$


\[
LV(ad) + size(ty) - 1 \leq v_2 \]

is valid. Then, we reach a new abstract state where the previous position \( p = (b_{\text{prep}}, b, i) \) is updated to the position \( p^+ = (b_{\text{prep}}, b, i + 1) \) of the next instruction in the same basic block, and we set \( LV(x) = w \) for a fresh \( w \in V_{sym} \). If we already know the value at the address \( ad \) (i.e., if there are \( w_1, w_2 \in V_{sym} \) with \( \models (a) \Rightarrow (LV(ad) = w_1) \) and \( w_1 \rightarrow_{ty} w_2 \in PT \)) then we add \( w = w_2 \) to \( KB \). Otherwise, we add \( LV(ad) \rightarrow_{ty} w \) to \( PT \). We used this rule to obtain \( B \) from \( A \) in Fig. 1. In a similar way, one can also formulate a rule for \( \text{store} \) instructions that store a value at some address in the memory (cf. [26]).

\[
\text{load from allocated memory} (p: "x = load ty* ad" with x, ad \in V_P) \]

\[
(p^+, LV \uplus \{x = v\}, KB, AL, PT) \quad \text{if} \quad (p^+, LV \uplus \{x = w\}, KB \cup \{w = w_2\}, AL, PT)
\]

- there is \( \text{alloc}(v_1, v_2) \in AL \) with \( \models (a) \Rightarrow (v_1 \leq LV(ad)) \wedge LV(ad) + size(ty) - 1 \leq v_2 \).
- there are \( w_1, w_2 \in V_{sym} \) with \( \models (a) \Rightarrow (LV(ad) = w_1) \) and \( w_1 \rightarrow_{ty} w_2 \in PT \).
- \( w \in V_{sym} \) is fresh

\[
(p^+, LV \uplus \{x = v\}, KB, AL, PT) \quad \text{if} \quad (p^+, LV \uplus \{x = w\}, KB \cup \{LV(ad) \leftarrow_{ty} w\})
\]

- there is \( \text{alloc}(v_1, v_2) \in AL \) with \( \models (a) \Rightarrow (v_1 \leq LV(ad) \wedge LV(ad) + size(ty) - 1 \leq v_2) \).
- there are no \( w_1, w_2 \in V_{sym} \) with \( \models (a) \Rightarrow (LV(ad) = w_1) \) and \( w_1 \rightarrow_{ty} w_2 \in PT \).
- \( w \in V_{sym} \) is fresh
If load accesses an address that was not allocated, then memory safety is violated and we reach the ERR state.

**load from unallocated memory** (p: “x = load ty* ad” with x, ad ∈ Vp)

\[
\begin{align*}
(p, LV, KB, AL, PT) & \quad \text{if there is no alloc(v1, v2) ∈ AL with } \models \langle a \rangle \Rightarrow (v_1 \leq LV(ad) \land LV(ad) + \text{size}(ty) - 1 \leq v_2) \\
\end{align*}
\]

The instructions icmp and br in strlen’s entry block check if the first character c0 is 0. In that case, we have reached the end of the string and jump to the block done. So for “x = icmp eq ty t1, t2”, we check if the state contains enough information to decide whether the values t1 and t2 of type ty are equal. In that case, the value 1 resp. 0 (i.e., true resp. false) is assigned to x.\(^\text{11}\)

**icmp** (p: “x = icmp eq ty t1, t2” with x ∈ Vp and t1, t2 ∈ Vp ∪ Z)

\[
\begin{align*}
(p^+, LV \uplus \{x = v\}, KB, AL, PT) & \quad \text{if } \models \langle a \rangle \Rightarrow (LV(t_1) = LV(t_2)) \\
(p^+, LV \uplus \{x = w\}, KB \uplus \{w = 1\}, AL, PT) & \quad \text{if } \models \langle a \rangle \Rightarrow (LV(t_1) = LV(t_2)) \\
(p^+, LV \uplus \{x = v\}, KB, AL, PT) & \quad \text{if } \models \langle a \rangle \Rightarrow (LV(t_1) \neq LV(t_2)) \\
(p^+, LV \uplus \{x = w\}, KB \uplus \{w = 0\}, AL, PT) & \quad \text{if } \models \langle a \rangle \Rightarrow (LV(t_1) \neq LV(t_2)) \\
\end{align*}
\]

The previous rule is only applicable if KB contains enough information to evaluate the condition. Otherwise, a case analysis needs to be performed, i.e., one has to refine the abstract state by extending its knowledge base. This is done by the following rule which transforms an abstract state into two new ones.\(^\text{12}\)

**refining abstract states** (p: “x = icmp eq ty t1, t2”, x ∈ Vp, t1, t2 ∈ Vp ∪ Z)

\[
\begin{align*}
(p, LV, KB, AL, PT) & \quad \text{if } \varphi \text{ is } LV(t_1) = LV(t_2) \text{ and both } \not\models \langle a \rangle \Rightarrow \varphi \quad \text{and} \quad \not\models \langle a \rangle \Rightarrow \neg \varphi
\end{align*}
\]

For example, in state B of Fig. 1, we evaluate “c0zero = icmp eq i8 c0, 0”, i.e., we check whether the first character c0 of the string str is 0. Since this cannot be inferred from B’s knowledge base, we refine B to the successor states C and D and call the edges from B to C and D refinement edges. In D, we have c0 = v1 and v1 ≠ 0. Thus, the icmp-rule yields E where c0zero = v2 and v2 = 0. We do not display the successors of C that lead to a program end.

The conditional branching instruction br is very similar to icmp. To evaluate “br i i t, label b_1, label b_2”, one has to check whether the current state contains enough information to conclude that t is 1 (i.e., true) or 0 (i.e., false). Then the evaluation continues with block b_1 resp. b_2. This rule allows us to create the successor F of E, where we jump to the block loop.

---

\(^{11}\) Other integer comparisons (for \(<, \leq, \ldots\)) are handled analogously.

\(^{12}\) Analogous refinement rules can also be used for other conditional LLVM instructions.
Since we do not consider the handling of data structures in this paper, we do not
readability, we again write “
the memory (i.e., that
$c$
load
To ensure memory safety, the
$M$
finitesimal
symbolic execution graphs, we
$M$
finite

\begin{align*}
\text{phi} & \; (p: \text{“} \phi x = \phi y [t_1, b_1], \ldots, [t_n, b_n] \text{” with } x \in V_p \cup \{0, 1\} \text{ and } b_1, b_2 \in \text{Blks}) \\
& \; \frac{(p, LV, KB, AL, PT)}{(b, b_1, 0), LV, KB, AL, PT} \quad \text{if } p = (b_{\text{prev}}, b, i) \text{ and } \models (\phi(x) = 1) \\
& \; \frac{(p, LV, KB, AL, PT)}{(b, b_2, 0), LV, KB, AL, PT} \quad \text{if } p = (b_{\text{prev}}, b, i) \text{ and } \models (\phi(x) = 0)
\end{align*}

Next, we have to evaluate a phi instruction. These instructions are needed
due to the static single assignment form of LLVM. Here, “$x = \phi ty [t_1, b_1], \ldots, [t_n, b_n]$” means that if the previous block was $b_j$, then the value $t_j$ is assigned
to $x$. All $t_1, \ldots, t_n$ must have type $ty$. Since we reached state $F$ in Fig. 1 after
evaluating the \textit{entry} block, we obtain the state $G$ with olds = $v_3$ and $v_3 = u_{\text{str}}$. The \texttt{strlen} function traverses the string using a pointer $s$ that is increased in
each iteration. The \texttt{loop} terminates, since eventually $s$ reaches the last memory
cell of the string (containing 0). Then one jumps to \texttt{done}, converts the pointers $s$ and \texttt{str} to integers, and returns their difference. To perform the required pointer
arithmetic, “$ad_2 = \text{getelementptr} ty \; ad_1, in \; t'$ increases $ad_1$ by the size of
$t$ elements of type $ty$ (i.e., by $\text{size}(ty) \cdot t$) and assigns this address to $ad_2$.\textsuperscript{13}

\begin{align*}
\text{getelementptr} & \; (p: \text{“} \text{getelementptr} \; ty \; ad_1, \text{in} \; t', \; ad_2 \in V_p, \; t \in V_p \cup \mathbb{Z}, \; b_1 \in \text{Blks}) \\
& \; \frac{(p, LV \uplus \{x = w\}, KB, AL, PT)}{(p', LV \uplus \{x = w\}, KB \cup \{v = LV(t')\}, \text{AL}, PT)} \quad \text{if } p = (b_j, b, k) \text{ and } \text{fresh}(w) \in V_{\text{sym}}
\end{align*}

In Fig. 1, this rule is used for the step from $G$ to $H$, where $LV$ and $KB$ now
imply $s = \text{str} + 1$. In the step to $I$, the character at address $s$ is loaded to $c$.
To ensure memory safety, the \texttt{load}-rule checks that $s$ is in an allocated part of the
memory (i.e., that $u_{\text{str}} \leq v_{\text{str}} + 1 \leq v_{\text{end}}$). This holds because $(H)$ implies
$u_{\text{str}} \leq v_{\text{end}}$ and $u_{\text{str}} \neq v_{\text{end}}$ (as $u_{\text{str}} \leadsto v_1, v_{\text{end}} \leadsto 0 \in PT$ and $v_1 \neq 0 \in KB$).
Finally, we check whether $c$ is 0. We again perform a refinement which yields
the states $J$ and $K$. State $K$ corresponds to the case $c \neq 0$ and thus, we obtain
czero = 0 in $L$ and branch back to instruction 0 of the \texttt{loop} block in state $M$.

\textbf{2.3 Generalizing Abstract States}

After reaching $M$, one unfolds the loop once more until one reaches a state
$\bar{M}$ at position $(\text{loop, loop, 0})$ again, analogous to the first iteration. To obtain
\textit{finite} symbolic execution graphs, we \textit{generalize} our states whenever an evaluation
visits a program position twice. Thus, we have to find a state that is more general
than $M = (p, LV_M, KB_M, AL, PT_M)$ and $M = (p, LV_{\bar{M}}, KB_{\bar{M}}, AL, PT_{\bar{M}})$. For
readability, we again write “$\leadsto$” instead of “$\leadsto_{18}$”. Then $p = (\text{loop, loop, 0})$ and

\textsuperscript{13} Since we do not consider the handling of data structures in this paper, we do not
regard \texttt{getelementptr} instructions with more than two parameters.
$$AL = \{ alloc(u_{\text{str}}, v_{\text{end}}) \}$$

$$LV_M = \{ \text{str} = u_{\text{str}}, c = v_5, s = v_4, \text{olds} = v_3, \ldots \}$$

$$LV_{\tilde{M}} = \{ \text{str} = u_{\text{str}}, c = \tilde{v}_5, s = \tilde{v}_4, \text{olds} = \tilde{v}_3, \ldots \}$$

$$PT_M = \{ u_{\text{str}} \leftrightarrow v_1, v_4 \leftrightarrow v_5, v_{\text{end}} \leftrightarrow z \}$$

$$PT_{\tilde{M}} = \{ u_{\text{str}} \leftrightarrow v_1, v_4 \leftrightarrow v_5, \tilde{v}_4 \leftrightarrow \tilde{v}_5, v_{\text{end}} \leftrightarrow z \}$$

$$KB_M = \{ v_3 \neq 0, v_4 = v_3 + 1, v_3 = u_{\text{str}}, v_1 \neq 0, z = 0, \ldots \}$$

$$KB_{\tilde{M}} = \{ \tilde{v}_3 \neq 0, \tilde{v}_4 = \tilde{v}_3 + 1, \tilde{v}_3 = v_1, v_4 = v_3 + 1, v_3 = u_{\text{str}}, v_1 \neq 0, z = 0, \ldots \}.$$

Our aim is to construct a new state $N$ that is more general than $M$ and $\tilde{M}$, but contains enough information for the remaining proof. We now present our heuristic for merging states that is used as the basis for our implementation.

To merge $M$ and $\tilde{M}$, we keep those constraints of $M$ that also hold in $\tilde{M}$. To this end, we proceed in two steps. First, we create a new state $N = (p, LV_N, KB_N, AL_N, PT_N)$ using fresh symbolic variables $v_x$ for all $x \in \mathcal{V}_p$ and define

$$LV_N = \{ \text{str} = v_{\text{str}}, c = v_5, s = v_4, \text{olds} = v_{\text{olds}}, \ldots \}.$$  

Matching $N$’s fresh variables to the variables in $M$ and $\tilde{M}$ yields mappings with $\mu_M (v_{\text{str}}) = u_{\text{str}}, \mu_M (v_c) = v_5, \mu_M (v_s) = v_4, \mu_M (v_{\text{olds}}) = v_3$, and $\mu_{\tilde{M}} (v_{\text{str}}) = u_{\text{str}}, \mu_{\tilde{M}} (v_c) = v_5, \mu_{\tilde{M}} (v_s) = v_4, \mu_{\tilde{M}} (v_{\text{olds}}) = v_3$. By injectivity of $LV_M$, we can also define a pseudo-inverse of $\mu_M$ that maps $M$’s variables to $N$ by setting $\mu^{-1}_M (LV_M(x)) = v_x$ for $x \in \mathcal{V}_p$ and $\mu^{-1}_M(v) = v$ for all other $v \in \mathcal{V}_{\text{sym}}$ ($\mu^{-1}_M$ works analogously).

In a second step, we use these mappings to check which constraints of $M$ also hold in $\tilde{M}$. So we set $AL_N = \mu^{-1}_M (AL) \cap \mu^{-1}_M (AL) = \{ alloc(v_{\text{str}}, v_{\text{end}}) \}$ and

$$PT_N = \mu^{-1}_M (PT_M) \cap \mu^{-1}_M (PT_{\tilde{M}})$$

$$= \{ v_{\text{str}} \leftrightarrow v_1, v_4 \leftrightarrow v_5, v_{\text{end}} \leftrightarrow z \} \cap \{ v_{\text{str}} \leftrightarrow v_1, v_4 \leftrightarrow v_5, v_4 \leftrightarrow v_5, v_{\text{end}} \leftrightarrow z \}$$

$$= \{ v_{\text{str}} \leftrightarrow v_1, v_4 \leftrightarrow v_5, v_{\text{end}} \leftrightarrow z \}.$$

It remains to construct $KB_N$. We have $v_3 = u_{\text{str}}$ (“olds = str”) in $\langle M \rangle$, but $\tilde{v}_3 = v_4, v_3 = v_3 + 1, v_3 = u_{\text{str}}$ (“olds = str + 1”) in $\langle \tilde{M} \rangle$. To keep as much information as possible in such cases, we rewrite equations to inequations before performing the generalization. For this, let $\langle M \rangle$ result from extending $\langle M \rangle$ by $t_1 \geq t_2$ and $t_1 \leq t_2$ for any equation $t_1 = t_2 \in \langle M \rangle$. So in our example, we obtain $v_3 \geq u_{\text{str}} \in \langle M \rangle$ (“olds $\geq$ str”). Moreover, for any $t_1 \neq t_2 \in \langle M \rangle$, we check whether $\langle M \rangle$ implies $t_1 > t_2$ or $t_1 < t_2$, and add the respective inequation to $\langle M \rangle$. In this way, one can express sequences of inequations $t_1 \neq t_2, t_1 + 1 \neq t_2, \ldots, t_1 + n \neq t_2$ (where $t_1 \leq t_2$) by a single inequation $t_1 + n < t_2$, which is needed for suitable generalizations afterwards. We use this to derive $v_4 < v_{\text{end}} \in \langle M \rangle$ (“$s < v_{\text{end}}$”) from $v_4 = v_3 + 1, v_3 = u_{\text{str}}, u_{\text{str}} \leq v_{\text{end}}, u_{\text{str}} \neq v_{\text{end}}$, $v_4 \neq v_{\text{end}} \in \langle M \rangle$.

We then let $KB_N$ consist of all formulas $\varphi$ from $\langle M \rangle$ that are also implied by $\langle \tilde{M} \rangle$, again translating variable names using $\mu^{-1}_M$ and $\mu^{-1}_{\tilde{M}}$. Thus, we have

$$\langle M \rangle = \{ v_3 \neq 0, v_4 = v_3 + 1, v_3 = u_{\text{str}}, v_4 \leq u_{\text{str}}, v_4 < v_{\text{end}}, \ldots \}$$

$$\mu^{-1}_M (\langle M \rangle) = \{ v_3 \neq 0, v_4 = v_{\text{olds}} + 1, v_{\text{olds}} = u_{\text{str}}, v_{\text{olds}} \leq v_{\text{str}}, v_4 < v_{\text{end}}, \ldots \}$$

$$\mu^{-1}_{\tilde{M}} (\langle \tilde{M} \rangle) = \{ v_4 \neq 0, v_4 = v_{\text{olds}} + 1, v_{\text{olds}} = v_4, v_4 = v_3 + 1, v_3 = u_{\text{str}}, v_4 < v_{\text{end}}, \ldots \}$$

$$KB_N = \{ v_4 \neq 0, v_4 = v_{\text{olds}} + 1, v_{\text{olds}} \geq v_{\text{str}}, v_4 < v_{\text{end}}, \ldots \}.$$
Definition 3 (Merging States). Let \( a = (p, LV_a, KB_a, AL_a, PT_a) \) and \( b = (p, LV_b, KB_b, AL_b, PT_b) \) be abstract states. Then \( c = (p, LV_c, KB_c, AL_c, PT_c) \) results from merging the states \( a \) and \( b \) if

- \( LV_c = \{ x = v_x \mid x \in \mathcal{V}_p \} \) for fresh pairwise different symbolic variables \( v_x \).
- Moreover, we define \( \mu_a(v_x) = LV_a(x) \) and \( \mu_b(v_x) = LV_b(x) \) for all \( x \in \mathcal{V}_p \) and let \( \mu_a \) and \( \mu_b \) be the identity on all remaining variables from \( \mathcal{V}_{sym} \).
- \( AL_c = \mu_a^{-1}(AL_a) \cap \mu_b^{-1}(AL_b) \) and \( PT_c = \mu_a^{-1}(PT_a) \cap \mu_b^{-1}(PT_b) \). Here, the “inverse” of the instantiation \( \mu_a \) is defined as \( \mu_a^{-1}(v) = v_x \) if \( v = LV_a(x) \) and \( \mu_a^{-1}(v) = v \) for all other \( v \in \mathcal{V}_{sym} \) (\( \mu_b^{-1} \) is defined analogously).
- \( KB_C = \{ \varphi \in \mu_a^{-1}(\langle a \rangle) \mid \models \mu_b^{-1}(\langle b \rangle) \supseteq \varphi \} \), where
  \[
  \langle a \rangle = (a) \cup \{ t_1 \geq t_2 \mid t_1 = t_2 \in \langle a \rangle \}
  \cup \{ t_1 > t_2 \mid t_1 \neq t_2 \in \langle a \rangle, \models \langle a \rangle \Rightarrow t_1 > t_2 \}
  \cup \{ t_1 < t_2 \mid t_1 \neq t_2 \in \langle a \rangle, \models \langle a \rangle \Rightarrow t_1 < t_2 \}.
  \]

In Fig. 1, we do not show the second loop unfolding from \( M \) to \( \widetilde{M} \), and directly draw a generalization edge from \( M \) to \( N \), depicted by a dashed arrow. Such an edge expresses that all concrete states represented by \( M \) are also represented by the more general state \( N \). Semantically, a state \( \pi \) is a generalization of a state \( a \) iff \( \models \langle a \rangle_{SL} \Rightarrow \mu(\langle \pi \rangle_{SL}) \) for some instantiation \( \mu \). To automate our procedure, we define a weaker relationship between \( a \) and \( \pi \). We say that \( \pi = (p, LV, KB, AL, PT) \) is a generalization of \( a = (p, LV, KB, AL, PT) \) with the instantiation \( \mu \) whenever the conditions (b)-(e) of the following rule are satisfied.

<table>
<thead>
<tr>
<th>Generalization with ( \mu )</th>
<th>( (p, LV, KB, AL, PT) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( a ) has an incoming evaluation edge (^{14})</td>
<td></td>
</tr>
<tr>
<td>(b) ( LV(x) = \mu(LV(x)) ) for all ( x \in \mathcal{V}_p ),</td>
<td></td>
</tr>
<tr>
<td>(c) ( \models \langle a \rangle \Rightarrow \mu(KB) ),</td>
<td></td>
</tr>
<tr>
<td>(d) if ( alloc(v_1, v_2) \in AL ), then ( alloc(\mu(v_1), \mu(v_2)) \in AL ),</td>
<td></td>
</tr>
<tr>
<td>(e) if ( (v_1 \leftrightarrow_{y} v_2) \in PT ), then ( (\mu(v_1) \leftrightarrow_{y} \mu(v_2)) \in PT )</td>
<td></td>
</tr>
</tbody>
</table>

Clearly, then we indeed have \( \models \langle a \rangle_{SL} \Rightarrow \mu(\langle \pi \rangle_{SL}) \). Condition (a) is needed to avoid cycles of refinement and generalization steps in the symbolic execution graph, which would not correspond to any computation.

Of course, many approaches are possible to compute such generalizations (or “widenings”). Thm. 4 shows that the merging heuristic from Def. 3 satisfies the conditions of the generalization rule. Thus, since \( N \) results from merging \( M \) and \( \widetilde{M} \), it is indeed a generalization of \( M \). Thm. 4 also shows that if one uses the merging heuristic to compute generalizations, then the construction of symbolic execution graphs always terminates when applying the following strategy:

- If there is a path from a state \( a \) to a state \( b \), where \( a \) and \( b \) are at the same program position, where \( b \) has an incoming evaluation edge, and where \( a \) has no incoming refinement edge, then we check whether \( a \) is a generalization of

\(^{14}\) Evaluation edges are edges that are not refinement or generalization edges.
b (i.e., whether the corresponding conditions of the generalization rule are satisfied). In that case, we draw a generalization edge from b to a.

- Otherwise, remove a’s children, and add a generalization edge from a to the merging c of a and b. If a already had an incoming generalization edge from some state q, then remove a and add a generalization edge from q to c instead.

**Theorem 4 (Soundness and Termination of Merging).** Let c result from merging the states a and b as in Def. 3. Then c is a generalization of a and b with the instantiations µa and µb, respectively. Moreover, if a is not already a generalization of b, then |ϕ| + |ALc| + |PTc| < |ϕa| + |ALa| + |PTa|. Here, for any conjunction ϕ, let |ϕ| denote the number of its conjuncts. Thus, the above strategy to construct symbolic execution graphs always terminates.\(^{15}\)

In our example, we continue symbolic execution in state N. Similar to the execution from F to M, after 6 steps another state P at position (loop, loop, 0) is reached. In Fig. 1, dotted arrows abbreviate several evaluation steps. As N is again a generalization of P using an instantiation µ with µ(vc) = w_c, µ(vs) = w_s, and µ(volds) = wolds, we draw a generalization edge from P to N. The construction of the symbolic execution graph is finished as soon as all its leaves correspond to ret instructions (for “return”).

Based on this construction, we now connect the symbolic execution graph to memory safety of the input program. We say that a concrete LLVM state (p, s, m) is represented by the symbolic execution graph iff the graph contains an abstract state a at position p where (s, m) = σ(⟨a⟩) for some concrete instantiation σ.

**Theorem 5 (Memory Safety of LLVM Programs).** Let P be an LLVM program with a symbolic execution graph G. If G does not contain the abstract state ERR, then P is memory safe for all LLVM states represented by G.

### 3 From Symbolic Execution Graphs to Integer Systems

To prove termination of the input program, we extract an integer transition system (ITS) from the symbolic execution graph and then use existing tools to prove its termination. The extraction step essentially restricts the information in abstract states to the integer constraints on symbolic variables. This conversion of memory-based arguments into integer arguments often suffices for the termination proof. The reason for considering only \(V_{sym}\) instead of \(V_P\) is that the conditions in the abstract states only concern the symbolic variables and therefore, these are usually the essential variables for proving termination.

For example, termination of strlen is proved by showing that the pointer s is increased as long as it is smaller than \(v_{end}\), the symbolic end of the input string. In Fig. 1, this is explicit since \(v_s < v_{end}\) is an invariant that holds in all states represented by N. Each iteration of the cycle increases the value of \(v_s\).

Formally, ITSs are graphs whose nodes are abstract states and whose edges are transitions. For any abstract state a, let \(V(a)\) denote the symbolic variables occurring in a. Let \(V \subseteq V_{sym}\) be the finite set of all symbolic variables occurring

\(^{15}\) The proofs for all theorems can be found in [26].
In the transition, we do not impose the additional constraints of \( P \) terminating, then only executed as long as \( v \). Here, \( V = \{ v' | v \in V \} \) represents the values of the variables after the transition. An ITS state \((a, \sigma)\) consists of an abstract state \( a \) and a concrete instantiation \( \sigma : V \rightarrow Z \). For any such \( \sigma \), let \( \sigma' : V' \rightarrow Z \) with \( \sigma'(v') = \sigma(v) \). Given an ITS \( \mathcal{I}, (a, \sigma) \) evaluates to \((\overline{a}, \overline{\sigma})\) (denoted \((a, \sigma) \rightarrow \mathcal{I} (\overline{a}, \overline{\sigma})\)) iff \( \mathcal{I} \) has a transition \((a, CON, \overline{\sigma})\) with \( \models (\overline{a} \cup \overline{\sigma}) \). Here, we have \((\sigma \cup \overline{\sigma})(v) = \sigma(v) \) \( (\sigma \cup \overline{\sigma})(v') = \overline{\sigma}(v') = \overline{\sigma}(v) \) for all \( v \in V \). An ITS \( \mathcal{I} \) is terminating iff \( \rightarrow \) is well founded.16

We convert symbolic execution graphs to ITSs by transforming every edge into a transition. If there is a generalization edge from \( a \) to \( \overline{a} \) with an instantiation \( \mu \), then the new value of any \( v \in V(\overline{\sigma}) \) in \( \overline{\sigma} \) is \( \mu(v) \). Hence, we create the transition \((a, (a) \cup \{ v' = \mu(v) | v \in V(\overline{\sigma}) \}, \overline{a})\).17 So for the edge from \( P \) to \( N \) in Fig. 1, we obtain the condition \( \{ w_s = w_{old} + 1, w_{old} = v_s, v_s < v_{end}, v'_{attr} = v_{attr}, v'_{end} = v_{end}, v'_c = v_c, v'_s = v_s, w_s, \ldots \} \). This can be simplified to \( \{ v_s < v_{end}, v'_{end} = v_{end}, v'_c = v_c, v_s = v_s + 1, \ldots \} \).

An evaluation or refinement edge from \( a \) to \( \overline{a} \) does not change the variables of \( V(a) \). Thus, we construct the transition \((a, (a) \cup \{ v' = v | v \in V(a) \}, \overline{a})\).

So in the ITS resulting from Fig. 1, the condition of the transition from \( A \) to \( B \) contains \( \{ v'_{end} = v_{end} \} \cup \{ u'_x = u_x | x \in V_B \} \). The condition for the transition from \( B \) to \( D \) is the same, but extended by \( v'_1 = v_1 \). Hence, in the transition from \( A \) to \( B \), the value of \( v_1 \) can change arbitrarily (since \( v_1 \notin V(A) \)), but in the transition from \( B \) to \( D \), the value of \( v_1 \) must remain the same.

**Definition 6 (ITS from Symbolic Execution Graph).** Let \( \mathcal{G} \) be a symbolic execution graph. Then the corresponding integer transition system \( \mathcal{I}_G \) has one transition for each edge in \( \mathcal{G} \):

- If the edge from \( a \) to \( \overline{a} \) is not a generalization edge, then \( \mathcal{I}_G \) has a transition from \( a \) to \( \overline{a} \) with the condition \( (a) \cup \{ v' = v | v \in V(a) \} \).
- If there is a generalization edge from \( a \) to \( \overline{a} \) with the instantiation \( \mu \), then \( \mathcal{I}_G \) has a transition from \( a \) to \( \overline{a} \) with the condition \((a) \cup \{ v' = \mu(v) | v \in V(\overline{\sigma}) \}\).

From the non-generalization edges on the path from \( N \) to \( P \) in Fig. 1, we obtain transitions whose conditions contain \( v'_{end} = v_{end} \) and \( v'_s = v_s \). So \( v_s \) is increased by 1 in the transition from \( P \) to \( N \) and it remains the same in all other transitions of the graph’s only cycle. Since the transition from \( P \) to \( N \) is only executed as long as \( v_s < v_{end} \) holds (where \( v_{end} \) is not changed by any transition), termination of the resulting ITS can easily be proved automatically.

The following theorem shows the soundness of our approach.

**Theorem 7 (Termination of LLVM Programs).** Let \( \mathcal{P} \) be an LLVM program with a symbolic execution graph \( \mathcal{G} \) that does not contain the state \( ERR \). If \( \mathcal{I}_G \) is terminating, then \( \mathcal{P} \) is also terminating for all LLVM states represented by \( \mathcal{G} \).

---

16 For programs starting in states represented by an abstract state \( a_0 \), it would suffice to prove termination of all \( \rightarrow \) -evaluations starting in ITS states of the form \( (a_0, \sigma) \).

17 In the transition, we do not impose the additional constraints of \( \overline{\sigma} \) on the post-variables \( V' \), since they are checked anyway in the next transition which starts in \( \overline{\sigma} \).
4 Related Work, Experiments, and Conclusion

We developed a new approach to prove memory safety and termination of C (resp. LLVM) programs with explicit pointer arithmetic and memory access. It relies on a representation of abstract program states which allows an easy automation of the rules for symbolic execution (by standard SMT solving). Moreover, this representation is suitable for generalizing abstract states and for generating integer transition systems. In this way, LLVM programs are translated fully automatically into ITSs amenable to automated termination analysis.

Previous methods and tools for termination analysis of imperative programs (e.g., AProVE [4, 5], ARMC [24], COSTA [1], Cyclist [7], FuncTion [29], Julia [25], KITTeL [12], LoopFrog [28], TAN [16], TRex [14], T2 [6], Ultimate [15], . . . ) either do not handle the heap at all, or support dynamic data structures by an abstraction to integers (e.g., to represent sizes or lengths) or to terms (representing finite unravelings). However, most tools fail when the control flow depends on explicit pointer arithmetic and on detailed information about the contents of addresses. While the general methodology of our approach was inspired by our previous work on termination of Java [4, 5], in the current paper we lift such techniques to prove termination and memory safety of programs with explicit pointer arithmetic. This requires a fundamentally new approach, since pointer arithmetic and memory allocation cannot be expressed in the Java-based techniques of [4, 5].

We implemented our technique in the termination prover AProVE using the SMT solvers Yices [11] and Z3 [20] in the back-end. A preliminary version of our implementation participated very successfully in the International Competition on Software Verification (SV-COMP) [27] at TACAS, which featured a category for termination of C programs for the first time in 2014. To evaluate AProVE’s power, we performed experiments on a collection of 208 C programs from several sources, including the SV-COMP 2014 termination category and standard string algorithms from [30] and the OpenBSD C library [23]. Of these 208 programs, 129 use pointers and 79 only operate on integers.

To prove termination of low-level C programs, one also has to ensure their memory safety. While there exist several tools to prove memory safety of C programs, many of them do not handle explicit byte-accurate pointer arithmetic (e.g., Thor [19] or SLayer [3]) or require the user to provide the needed loop invariants (as in the Jessie plug-in of Frama-C [21]). In contrast, our approach can prove memory safety of such algorithms fully automatically. Although our approach is targeted toward termination and only analyzes memory safety as a prerequisite for termination, it turned out that on our collection, AProVE is more powerful than the leading publicly available tools for proving memory safety. To this end, we compared AProVE with the tools CPAchecker [18] and Predator [10] which reached the first and the third place in the category for memory safety at SV-COMP 2014. For the 129 pointer programs in our collection, AProVE can show memory safety for 102 examples, whereas CPAchecker resp. Predator prove

\[ 18 \] The second place in this category was reached by the bounded model checker LLBMC [13]. However, in general such tools only disprove, but cannot verify memory safety.
memory safety for 77 resp. 79 examples (see [2] for details).

To evaluate the power of our approach for proving termination, we compared AProVE to the other tools from the termination category of SV-COMP 2014. In addition, we included the termination analyzer KITTeL [12] in our evaluation, which operates on LLVM as well. On the side, we show the performance of the tools on integer and pointer programs when using a time limit of 300 seconds for each example. Here, we used an Intel Core i7-950 processor and 6 GB of memory. “T” gives the number of examples where termination could be proved, “N” is the number of examples where non-termination could be shown, “F” states how often the tool failed in less than 300 seconds, “TO” gives the number of time-outs (i.e., examples for which the tool took longer than 300 seconds), and “RT” is the average runtime in seconds for those examples where the tool proved termination or non-termination. For pointer programs, we omitted the results for those tools that were not able to prove termination of any examples.

Most other termination provers ignore the problem of memory safety and just prove termination under the assumption that the program is memory safe. So they may also return “Yes” for memory unsafe programs and may treat read accesses to the heap as non-deterministic input. Since AProVE constructs symbolic execution graphs to prove memory safety and to infer suitable invariants needed for termination proofs, its runtime is often higher than that of other tools. On the other hand, the table shows that our approach is slightly more powerful than the other tools for integer programs (i.e., our graph-based technique is also suitable for programs on integers) and it is clearly the most powerful one for pointer programs. The reason is due to our novel representation of the memory which handles pointer arithmetic and keeps information about the contents of addresses. For details on our experiments and to access our implementation in AProVE via a web interface, we refer to [2]. In future work, we plan to extend our approach to recursive programs and to inductive data structures defined via struct (e.g., by integrating existing shape analyses based on separation logic).

Acknowledgments. We are grateful to the developers of the other tools for termination or memory safety [6, 10, 12, 15, 16, 18, 29] for their help with the experiments.

References
2. AProVE: http://aprove.informatik.rwth-aachen.de/eval/Pointer/
27. SV-COMP at TACAS 2014: http://sv-comp.sosy-lab.org/2014/