Bounded Verification and Testing of Heap Programs

Duc-Hiep Chu
IST Austria
Email: duc-hiep.chu@ist.ac.at

Joxan Jaffar and Andrew E. Santosa
National University of Singapore
Email: {joxan@comp.,dcsandr@}nus.edu.sg

***Abstract*—We present an algorithm and implementation for the bounded verification or testing of heap-manipulating programs in pursuit of safety properties. This algorithm is based on symbolic execution whose exploration covers every execution path up to a certain length. The novel feature is the use of symbolic heaps in order to precisely model the effect of dynamic memory allocation, and critically, the use of property-directed *interpolation* in order to obtain a useful relaxation of the path constraints. In other words, we perform dynamic symbolic execution with *pruning*. Finally, we describe our implementation TRACER-X and present an experimental evaluation against LLBMC, a state-of-the-art system for bounded verification, and KLEE, a state-of-the-art dynamic symbolic executor used for testing.**

I. Introduction

Symbolic execution (SE) has emerged as a important method to reason about programs, in both verification and testing. By reasoning about inputs as symbolic entities, its fundamental advantage over traditional testing, which uses concrete inputs, is simply that it has better *coverage* of *program paths*. In particular, *dynamic symbolic execution* (DSE), where the execution space is explored *path-by-path*, has been shown effective in systems such as DART [1], CUTE [2], KLEE [3].

A key advantage of DSE is that by examining a single path, the analysis can be both precise (for example, capturing intricate details such as the state of the cache micro-architecture), and efficient (for example, the constraint solver often needs to deal with path constraints that are a single disjunction). Another advantage is the possibility of reasoning about system or library functions which we can execute but not analyze, as in the method of *concolic testing*. Yet another advantage is the ability to realize a *search strategy* in the path exploration, for example, to perform in random manner, or a depth/breadth-first manner, or in a manner determined by the program structure. However, the key disadvantage of DSE is that the number of program paths is in general exponential in the program size, and most available implementations of DSE do not employ a general technique to prune away some paths. Indeed, a recent paper [4] describes that DSE "traditionally forks off two executors at the same line, which remain subsequently forever independent", clearly suggesting that the DSE processing of different paths have no symbiosis.

The counterpart to DSE may be called *static symbolic execution* (SSE), or *bounded model checking* (BMC) where, typically, the (whole) program is encoded by a constraint solver, typically an SMT solver [5]. Now because the encoding is manageable in size (typically not more than quadratically bigger than the program, this apparently addresses the key disadvantage of DSE of having exponentially many paths to explore. However, the exploration process is merely delegated away to the solver, which then has to deal with the general problem of reasoning about what is essentially a huge disjunction. Nevertheless, constraint solvers, which routinely deal with intractable problems, have the *opportunity* of solving such problems. A most notable feature of constraint solvers is that of *clause learning* which can enable efficient solution of intractable problems. Essentially, clause learning enables "pruning"in exploration process of the solver. Some notable BMC systems are CBMC [6] and LLBMC [7].

In this paper, we consider an approach which is different from traditional DSE and SSE. We consider the method of *abstraction learning* [8], which is more popularly known as *lazy annotations* (LA) [9], [10]. This method has been implemented in the TRACER [11], [12] which was among the first systems to demonstrate DSE with pruning. While TRACER was able to perform bounded verification and testing on many examples, it could not accommodate industrial programs. Instead, it was primarily used to evaluate new algorithms in verification, analysis and testing, e.g., [13], [14], [15]. It was not intended to be a stand-alone verifier or tester.

In this paper, we extend the algorithm of TRACER. The main new feature we contribute here is in constraint reasoning about *dynamically allocated/freed* memory. The technical approach involves a logical interpretation of dynamically allocated (but not yet freed) addresses as *symbolic* addresses. This interpretation is on the one hand *abstract*, for because dynamically created addresses can be reasoned about in an existential manner, and on the other hand, *precise*, because it captures the desired collection of concrete states. More specifically, we develop a logic formula that precisely captures the notion of subsumption of states containing symbolic addresses, and then develop a method of propagating these formulas so that they can be used for pruning the exploration space. This contribution may be utilized in any symbolic execution framework that wishes to reason about different execution paths, e.g. KLEE [3], DSM [16], and even Veritesting [4].

A follow-up contribution is to demonstrate an implementation of symbolic reasoning in an LA system TRACER-X, built on top of KLEE. TRACER-X is a successor to the LA system TRACER, but now accommodates heap-manipulating programs. We are unaware of any symbolic execution system which reasons about dynamic memory with both the precision that captures all the required information, and yet with an

```
⟨1⟩ #define MAX 30
    n = input(); // getting a symbolic input
    x = malloc(sizeof(int)); *x = n;
    for (int i = 0; i < MAX; i++) {
⟨2⟩    if (*) { y = malloc(sizeof(int)); *y = *x+1; }
       else { y = malloc(sizeof(int)); *y = *x+1; } //(#)
       x = y;
    }
⟨3⟩ assert(*y >= MAX + n);
```

Fig. 1: Motivating Example

abstraction level that permits pruning. We conclude with an experimental evaluation comparing with LLBMC, state-of-the-art BMC tool, and KLEE, a state-of-the-art DSE tool for testing memory errors.

II. Motivating Examples

We begin with the example in Figure 1, and the observation that its symbolic execution times out (> 1 hour) using both KLEE and LLBMC. Clearly the number of symbolic execution paths is 2^{MAX}. Thus for a "non-pruning" system like KLEE [3], the reason for timeout is obvious. However, LLBMC [7], which processes (a manageable sized encoding of all) the paths by using an SMT solver, we observe that despite the pruning capability of SMT, this example still could not be fully executed.

To see the core problem, first note that every path is *essentially the same*. Let us hand execute this example, with a reduced MAX = 2. Consider any two paths, for example, in one, we (always) take the "then" alternative in each branch, and in the other, we always take the "else" alternative. At the end of the first path, the heap and stack, described as two collections of key-value pairs, look like

$$\begin{array}{l} heap: x_0 \mapsto n_0, y_1 \mapsto n_0+1, y_2 \mapsto n_0+2 \\ stack: n \mapsto n_0, x \mapsto y_2, y \mapsto y_2 \end{array} \tag{1}$$

where n_0 represents the symbolic input, x_0 represents the address of the first memory allocation, and y_1, y_2 represent the two addresses allocated in the `for` loop. Similarly, at the end of the second path, we have

$$\begin{array}{l} heap: x_0 \mapsto n_0, z_1 \mapsto n_0+1, z_2 \mapsto n_0+2 \\ stack: n \mapsto n_0, x \mapsto z_2, y \mapsto z_2 \end{array} \tag{2}$$

where z_1, z_2 represent the two addresses of the dynamically allocated memory by the loop along this path. The crux of the problem is to determine that these two state configurations (1) and (2) are the same. In the case of DSE such as KLEE [3], DSM [16] (aka. KLEE with state merging), and Veritesting [4], concrete values are assigned to y_1, y_2, and different concrete values are assigned to z_1, z_2. Thus the two state configurations must be consider as *different*. In the case of LLBMC, *statically defined* segmented arrays are used for SMT-encoding of the program. Thus the two paths in consideration will involve different (SMT) variables. Conflict clause learning thus does not work across different (statically defined) arrays.

Our main contribution is to regard the dynamically allocated addresses symbolically. What this means is essentially that, in the logical reading of a symbolic state, we regard these addresses, as well as symbolic inputs, as *existentially quantified* variables. That is, (1) should read as the following formula[1] with free variables n, x, and y:

$$\exists n_0 \exists x_0 \exists y_1 \exists y_2 \left(\begin{array}{l} x_0 \mapsto n_0, y_1 \mapsto n_0+1, y_2 \mapsto n_0+2 \\ n \mapsto n_0, x \mapsto y_2, y \mapsto y_2 \end{array} \right)$$

With this relaxation, the state configurations (1) and (2) are considered the same and need not be both explored. Consequently, TRACER-X can run this example quickly. (In fact in near linear time because our argument applies inductively, thus the number of nodes in the symbolic execution tree that we actually encounter grows linearly with MAX.)

Now consider a small change to the example in Figure 1 where in the line marked (#), the increment value is 2 instead of 1. This time our two paths will produce two different heaps (their contents are different). Thus to be efficient, we also need the power of abstraction learning. Our algorithm, which is based upon a path-by-path reasoning process, is able to produce an *interpolant* which describes a relaxation of the symbolic state based upon the target property to be proved. In other words, our algorithm is *property-directed* and able to perform a kind of weakest precondition. We relegate the details to Section IV. Meanwhile, we simply mention that for this example, while both KLEE and LLBMC take exponential time, our algorithm executes in near linear time.

Finally, in Figure 2, we display a real-world benchmark examples, which checks if a string matches a regular expressions (and is tutorial example 2 in the KLEE distribution). Note similarity of this program with Figure 1. Here, code iterations are in the form of recursive function calls, and dynamic memory arises from the instances of the formal function parameters. The content of this memory are addresses of statically allocated arrays plus some offset. Different symbolic execution paths arise from different conditions that the symbolic arrays `re[]` and `text[]` satisfy, and these variations lead to recursive calls that increment the values of pointer values by various offsets (0, 1 or 2, in this example). The objective is to show that no reference is made outside the array footprints. Unsurprisingly, here too we observe that both KLEE and LLBMC time out while TRACER-X runs quickly; section VI provides some run-time numbers.

III. Symbolic Execution of Heap Programs

We formalize DSE (as in KLEE-like systems [3], [16], [4]) for a subset language of LLVM. Though being simple, the language is enough to demonstrate the difficulties to apply "lazy annotation" when dealing with heap-based programs.

We model a program P by a transition system: a tuple $\langle \Sigma, \ell_{\mathsf{start}}, \longrightarrow \rangle$ where Σ is the set of program points and $\ell_{\mathsf{start}} \in \Sigma$ is the *unique* initial program point. Let $\longrightarrow \subseteq \Sigma \times \Sigma \times Stmts$,

[1] Strictly speaking, we also need to enforce the "separation" between the allocated chunks of memory. This topic cannot be easily discussed without proper formalization first.

```
int matchstar(int c, char *re, char *text) {
  do if (matchhere(re, text)) return 1;
  while (*text!='\0' && (*text++==c || c=='.'));
  return 0;
}

int matchhere(char *re, char *text) {
  if (re[0] == '\0') return 0;
  if (re[1] == '*') return matchstar(re[0], re+2, text);
  if (re[0] == '$' && re[1]=='\0') return *text == '\0';
  if (*text!='\0'&&(re[0]=='.'||re[0]==*text))
      return matchhere(re+1, text+1);
  return 0;
}

int match(char *re, char *text) {
  if (re[0] == '^') return matchhere(re+1, text);
  do if (matchhere(re, text)) return 1;
  while (*text++ != '\0');
  return 0;
}

#define SIZE 7
main() {
    char re[SIZE]; re[SIZE-1] = '\0';
    match(re, "hello");
}
```

Fig. 2: Small but Real World Example

where *Stmts* is the set of program statements, be the transition relation that relates a state to its (possible) successors by executing the statements. We shall use $\ell \xrightarrow{\mathsf{stmt}} \ell'$ to denote a transition relation from $\ell \in \Sigma$ to $\ell' \in \Sigma$ executing the statement `stmt` $\in$ *Stmts*.

Let the program variables be denoted by *Vars*. Basic statements are defined in Table I. Note that "assertions" can be modeled with "assume" and "error" statements. Other than the program variables, there are also *logical* variables. Let $\tilde{I}$ denote the symbolic inputs, which are *logical* variables.

stmt	::=	*var* := input() | *var* := *exp* | store(*exp*, *exp*) | assume(*exp*) | *var* := *malloc*(c) | error | halt
exp	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | c
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
c	::=	32-bit unsigned integer
var $\in$ *Vars*		

TABLE I
A SIMPLE INTERMEDIATE LANGUAGE

For presentation purposes, we consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values, and memory objects are aligned in 4-bytes. Generalizing to additional types is straightforward. We also assume that the input programs are well-typed in the obvious way.

We now define the notion of symbolic state. In addition to a (stack) store in traditional definition (e.g. in [12]), we also track a (symbolic) "heap store".

Definition 1 (Symbolic State): A symbolic state s is a tuple $\langle \ell, \sigma, \mu, \Pi \rangle$, where $\ell \in \Sigma$ is the current program point, the *store* σ is a map from program variables to terms, the *heap store* μ is a map from addresses to terms, and the path condition Π is a first-order formula over the symbolic inputs that the inputs must satisfy in order for the execution to reach the current state. $\square$

Let Ω denote an *empty* map. Given a map M, let $dom(M)$ be the domain of M. From Definition 1, for every symbolic state $\langle \ell, \sigma, \mu, \Pi \rangle$, $Vars \equiv dom(\sigma)$. Let $M[key \mapsto val]$ denote the most basic operation of a map: if *key* does not exist in M, it will be added; and then the associated value for *key* in M is set to *val*. We also define the binary relation $\sqsubseteq$ between maps as follows: $M_1 \sqsubseteq M_2$ if $dom(M_1) \subseteq dom(M_2)$ and $\forall x \in dom(M_1) \cdot M_1(x) \equiv M_2(x)$.

The *evaluation* $[\![e]\!]^{\mu}_{\sigma}$ of an expression e with the stores σ and μ is defined in a standard way. (Note that our expressions have no *side-effects*. Calls to system functions must be treated differently, such as in the case of `malloc`.) For example, $[\![v]\!]^{\mu}_{\sigma} = \sigma(v)$ (if v is a program variable), $[\![c]\!]^{\mu}_{\sigma} = c$ (if c is an integer), $[\![e_1 \Diamond_b e_2']\!]^{\mu}_{\sigma} = [\![e_1]\!]^{\mu}_{\sigma} \Diamond_b [\![e_2]\!]^{\mu}_{\sigma}$ (where e_1, e_2 are expressions and $\Diamond_b$ is a binary operator), and $[\![load(e)]\!]^{\mu}_{\sigma} = \mu([\![e]\!]^{\mu}_{\sigma})$. The notion of evaluation is extended for a set of constraints in an intuitive way.

A symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi \rangle$ is called *infeasible* if Π is unsatisfiable. Otherwise, the state is called *feasible*; symbolic execution is possible from a feasible state only.

Definition 2 (Transition Step): Given a transition system $\langle \Sigma, \ell_{\mathsf{start}}, \longrightarrow \rangle$ and a feasible symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi \rangle$, the symbolic execution of transition $\ell \xrightarrow{\mathsf{stmt}} \ell'$ returns a *successor state* $\langle \ell', \sigma', \mu', \Pi' \rangle$ where σ', μ', Π' are computed as in Table II. $\square$

`stmt`	σ'	μ'	Π'
v := input()	$\sigma[v \mapsto i]$	μ where i is a fresh (logical) variable	Π
v := e	$\sigma[v \mapsto [\![e]\!]^{\mu}_{\sigma}]$	μ	Π
store(e_1, e_2)	σ	$\mu[[\![e_1]\!]^{\mu}_{\sigma} \mapsto [\![e_2]\!]^{\mu}_{\sigma}]$	Π
assume(e)	σ	μ	$\Pi \wedge [\![e]\!]^{\mu}_{\sigma}$
v := malloc(c)	$\sigma[v \mapsto a]$	$\mu[a \mapsto \_] \cdots [a+c/4-1 \mapsto \_]$ where a is the address returned from concretely executing malloc(c)	Π

TABLE II
OPERATIONAL SEMANTICS FOR SYMBOLIC EXECUTION

Let $s_0 \equiv \langle \ell_{\mathsf{start}}, \Omega, \Omega, true \rangle$ be the *initial* symbolic state. A *symbolic path* $s_0 \rightarrow s_1 \cdots \rightarrow s_m$ is a sequence of symbolic states such that $\forall 1 \leq i \leq m$, s_i is a *successor* of s_{i-1}.

We can now define *symbolic exploration* as the process of constructing a *Symbolic Execution Tree* (SET) rooted at s_0. In accordance to some *search strategy*, the order in which the nodes are constructed can be different. For bounded verification and testing, we assume the tree depth is bounded.

The reachability of an `error` statement indicates a *bug*. Symbolic execution typically stops the path and generates a *failed* test case witnessing that bug. On the other hand, a path *safely* terminates if we reach a `halt` statement, and we also generate a *passed* test case. We prove a program is *safe* by showing that no `error` statement is reached. A subtree is called *safe* if no `error` statement is reached from its root.

IV. Abstraction Learning via Interpolation

Naively enumerating the Symbolic Execution Tree (SET) will not scale. In symbolic exploration, we would like to prune away those states that guarantee to *not* lead to "any unknown error". We discuss simple state subsumption in Section IV-A first, then we will discuss interpolation in Section IV-B. Section IV-C shows that a straightforward adaptation of "lazy annotation" does not work. Our solution will be presented afterward.

A. Subsumption between States

Definition 3 (Concrete State): A concrete state cs is a tuple $\langle \ell, S, H\rangle$, where $\ell \in \Sigma$ is the current program point, the stack S is a map from program variables to (concrete) values, and the *heap* H is a map from addresses to values. □

Definition 4 (Subsumption of Concrete States): Given two concrete states $cs_1 \equiv \langle \ell, S_1, H_1\rangle$ and $cs_2 \equiv \langle \ell, S_2, H_2\rangle$, we say cs_1 *subsumes* cs_2 if $S_1 \sqsubseteq S_2$ and $H_1 \sqsubseteq H_2$. □

The implication of "cs_1 subsumes cs_2" is that if the execution of a program fragment starting from cs_1 is error-free, then the executing the same fragment starting from cs_2 will also be error-free. (Another interpretation: the execution with cs_1 contains more *undefined* and *erroneous* behaviors than the execution with cs_2.) We also remark that because $Vars \equiv dom(S_1) \equiv dom(S_2)$, $S_1 \sqsubseteq S_2$ simply means $S_1 \equiv S_2$.

Given a feasible symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi\rangle$ that is reachable from s_0 via a sequence of transitions π, let θ be a satisfying assignment of symbolic variables $\tilde{I}$ for the path condition Π, i.e. $[\![\Pi]\!]_\theta \equiv true$. Let $[\![s]\!]_\theta$ denote the concrete state $cs \equiv \langle \ell, S, H\rangle$, where $S \equiv [\![\sigma]\!]_\theta$ and $H \equiv [\![\mu]\!]_\theta$. Then cs is the concrete state resulting from executing the state $[\![s_0]\!]_\theta$ (via the same sequence of transitions π).

We are now ready to define the notion of subsumption for two symbolic states.

Definition 5 (Subsumption of Symbolic States): Given two symbolic states $s_1 \equiv \langle \ell, \sigma_1, \mu_1, \Pi_1\rangle$ and $s_2 \equiv \langle \ell, \sigma_2, \mu_2, \Pi_2\rangle$, we say s_1 *subsumes* s_2, denoted by $s_2 \models s_1$, if for each satisfying assignment θ_2 of Π_2, there exists a satisfying assignment θ_1 of Π_1 such that $[\![s_1]\!]_{\theta_1}$ *subsume* $[\![s_2]\!]_{\theta_2}$. □

Of course in practice we will not enumerate all the satisfying assignments for Π_2 and Π_1 to perform the subsumption check. Instead, we make use of a symbolic solver. Let $\tilde{I}_1$ and $\tilde{I}_2$ be the symbolic variables denote the inputs for s_1 and s_2 respectively, then the subsumption check is *equivalent* to the validity of the following formula:

$$\forall \tilde{I}_2 \left[\Pi_2 \Rightarrow \exists \tilde{I}_1 \begin{pmatrix} \Pi_1 \wedge \\ dom(\sigma_1) \subseteq dom(\sigma_2) \wedge \\ dom(\mu_1) \subseteq dom(\mu_2) \wedge \\ \forall x \in dom(\sigma_1) \cdot (\sigma_1(x) = \sigma_2(x)) \wedge \\ \forall a \in dom(\mu_1) \cdot (\mu_1(a) = \mu_2(a)) \end{pmatrix} \right] \quad (3)$$

By construction we always have $dom(\sigma_2) \equiv Vars$ which by default will cover the store in s_1, thus the constraint $dom(\sigma_1) \subseteq dom(\sigma_2)$ can be dropped.

Proposition 1: The subsumption condition in Definition 5 and the validity of the formula (3) are *equivalent*. □

B. Interpolation for Verification vs. Testing

Relying on state subsumption alone is not enough. It is because different symbolic states that share the same program point might be significantly different, e.g., due to different increments as in the modified version of Figure 1. On the other hand, abstraction learning, has demonstrated significant speedup in verification and testing, e.g., [8], [9], [12], [15]. The intuitive idea is as follows.

In exploring the SET, an interpolant Ψ of a state s is an *abstraction* of it, which ensures the safety of the subtree rooted at that state. In other words, if we continue the execution with Ψ instead of s, we will *not* reach any error.

Upon encountering a state $\bar{s}$ of the same program point as s, i.e., s and $\bar{s}$ have same set of emanating transitions, if $\bar{s} \models \Psi$, then continuing the execution from $\bar{s}$ will not lead to any error. Consequently, we can prune the subtree rooted at $\bar{s}$.

Interpolants are computed recursively from bottom up. We now describe the form of our interpolants and the *base cases*.

Suppose we have a symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi\rangle$. In this paper, an interpolant Ψ for s is of the same form as a symbolic state. That is, we compute $\Psi \equiv \langle \ell, \bar{\sigma}, \bar{\mu}, \bar{\Pi}\rangle$ where $\bar{\sigma} \sqsubseteq \sigma$, $\bar{\mu} \sqsubseteq \mu$, and $\Pi \implies \bar{\Pi}$, yet Ψ satisfies the requirement that it ensures the safety of the subtree rooted at s. It is important to note that the check if an interpolant subsumes a symbolic state, can simply follow the Definition 5 and formula (3).

Safely terminated state: If s halts normally, the interpolant is simply $\langle \ell, \Omega, \Omega, true\rangle$.

Infeasible state: Given the transition $t \equiv \ell \xrightarrow{\texttt{stmt}} \ell'$ executing it from s results in a successor state s' that is infeasible. (`stmt` must be an `assume` statement.) Then $\langle pc', \Omega, \Omega, false\rangle$ is an interpolant of s'.

Error state: The treatment of error states differs for verification and testing. When an error statement is reached, the verification task fails and we will generate a test case witnessing the bug and then terminate the exploration process. However, in testing (and test case generation), we cannot simply stop at the first error path. For the purpose of this paper, the interpolant for an error state is simply $\langle \ell, \Omega, \Omega, true\rangle$. The consequence is that we will not generate error paths duplicating the same error location. (This is what typical testers such as KLEE will do, because in general there could be exponentially many paths leading to the same error location.)

Interpolants are propagated backward in the same manner as how the "weakest" precondition is computed. More formally, let $\mathsf{pre}(t, \Phi)$ denote the precondition of the transition t wrt. the postcondition Φ, then the interpolant Ψ of state s is computed as follows:

```
Ψ := true
foreach successor s_i of s wrt. transition t_i
   Let Ψ_i be the computed interpolant for s_i
   Ψ := Ψ ∧ pre(t_i, Ψ_i)
```

We elaborate on our implementation of $\mathsf{pre}(\_,\_)$ in Section V.

C. Interpolation with Heaps

Let us revisit the first motivating example in Figure 1 (with $MAX = 30$). Let $s_1 \equiv \langle \ell, \sigma_1, \mu_1, \Pi_1 \rangle$ and $s_2 \equiv \langle \ell, \sigma_2, \mu_2, \Pi_2 \rangle$ be two symbolic states at program point $\langle 3 \rangle$, i.e., right before the assertion. Also assume that we visit s_1 before s_2 in our exploration.

Even though s_1 and s_2 are very much similar, subsumption cannot happen because `malloc` returned different sequence of addresses in the two paths, thus the domains of μ_1 and μ_2 are different. In other words, there do not exist θ_1 and θ_2 such that $[\![\mu_1]\!]_{\theta_1} \sqsubseteq [\![\mu_2]\!]_{\theta_2}$.

If interpolation is enabled, let n_0 be the symbolic input (that n holds) and a be a concrete address returned by the last `malloc`, then an interpolant for s_1 can be:

$$\Psi \equiv \langle \ell, \{n \mapsto n_0, y \mapsto a\}, \{a \mapsto n_0 + 30\}, true \rangle.$$

For the same reason, this interpolant will not subsume s_2 either. That is, $s_2 \models \Psi$ does not hold. In particular, in s_2, y is mapped to a different address, returned by a different call to `malloc`, thus $dom(\mu_2)$ contains that address instead of a.

We now present our new treatment for malloc, which is foundational to enable heap interpolation. Essentially, the semantics of `malloc` is precisely captured using *symbolic* addresses.

Definition 6 (Transition Step for `malloc`*):* Given a feasible symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi \rangle$, the symbolic execution of transition $\ell \xrightarrow{\texttt{malloc(c)}} \ell'$ returns a *successor state* $\langle \ell', \sigma', \mu', \Pi' \rangle$ where σ', μ', Π' are computed as follows:

- σ' := $\sigma[v \mapsto a]$
- μ' := $\mu[a \mapsto \_] \ \ldots \ [a + c/4 - 1 \mapsto \_]$
- Π' := $\Pi \wedge dom(\mu') = [a, a + c/4 - 1] \uplus dom(\mu)$

where a is a *fresh symbolic variable*. □

The key change is that instead of using a concrete address returned by a system call to `malloc`, we use a fresh symbolic variable. We also add into the path condition the constraints specifying that the newly-allocated region is separated from the domain of the old heap store μ and the new domain of the new heap store include both of them. We use the notation $\uplus$ to succinctly represent these constraints.

Side Remarks: Firstly, separation constraints of m allocated "regions", denoted by $[a_1, b_1] \ldots [a_m, b_m]$, can be easily encoded by simple arithmetic constraints such as $a_i > b_j \vee a_j > b_i$. The size of such a formula is *quadratic* to the number of allocations in a path. Secondly, to be faithful to possible side-effects, we also invoke `malloc` and record the returned address value. Yet the value is neither used for interpolating nor subsumption checking.

Note that our new treatment for `malloc` introduces another set of logical variables, denoted by $\tilde{A}$, representing the symbolic addresses capturing what were returned from calling `malloc`. So in the check of whether an interpolant subsumes a symbolic state using formula (3), the set of quantified variables also includes $\tilde{A}$. We relegate to Section V to discuss how we implement such check using a standard solver.

We now conclude the section with a formal statement about our pruning using interpolation.

Theorem 1: Given a symbolic state $\bar{s}$ and an interpolant Ψ such that $\bar{s} \models \Psi$, then the expansion of $\bar{s}$ will not lead to any (unknown) errors, i.e. pruning $\bar{s}$ is *sound*. □

V. TRACER-X: Design and Implementation

Our system TRACER-X [17] combines TRACER [11], [12] and KLEE [3]. KLEE is a well-developed DSE tool, and TRACER was a test bed for path pruning in DSE using the lazy annotation method. The first goal of TRACER was to have both DSE and pruning. The current implementation TRACER-X first improved TRACER by building on top of KLEE, thereby inheriting its C++/C and LLVM applicability, and its efficiency for DSE. The main contribution of TRACER-X however is to implement symbolic states which accurately describe dynamically changing heaps, and an interpolation algorithm for reasoning about such states.

Reusing the infrastructure of KLEE implies that our symbolic execution needs to track more details, thus going beyond Section III, whose purpose was to present the background for symbolic execution when heap memory is involved. For example, to detect *buffer overflow* errors, each pointer variable is also associated to a base address of an allocated memory region that it is supposed to point to. Assignment of pointers will also pass such base address around. Buffer overflow is detect if we dereference a pointer whose value goes over (or under) the addresses of the associated region.

In the following, we will focus on the two key implementation features, which concern the how interpolants are propagated backward and the subsumption check. Finally, we will illustrate using the motivating example in Figure 1.

A. Backward Propagation of Interpolants

We have presented the base cases in Section IV, here we describe how an interpolant is *propagated* from a child node to its parent. Suppose executing a transition $t \equiv \ell \xrightarrow{\texttt{stmt}} \ell'$ from a symbolic state $s \equiv \langle \ell, \sigma, \mu, \Pi \rangle$ results in state s'.

We start with the case where s' is infeasible. Thus `stmt` must be an assume statement, denoted by `assume(e)`. Then $\langle pc', \Omega, \Omega, false \rangle$ is an interpolant of s'.

Recalling how an `assume` statement is executed, we have $s' \equiv \langle \ell', \sigma, \mu, \Pi \wedge [\![e]\!]^{\mu}_{\sigma} \rangle$. So we can compute $\langle \ell, \bar{\sigma}, \bar{\mu}, \bar{\Pi} \rangle \equiv \mathsf{pre}(t, \langle pc', \Omega, \Omega, false \rangle)$, as follows:

- $\bar{\Pi}$ is the first-order interpolant as in [9], [8] such that $\Pi \implies \bar{\Pi}$ and $\bar{\Pi} \wedge [\![e]\!]^{\mu}_{\sigma} \implies false$; we compute $\bar{\Pi}$ simply from the *unsatisfiability core* of the infeasibility proof of $\Pi \wedge [\![e]\!]^{\mu}_{\sigma}$.
- $\bar{\sigma} \sqsubseteq \sigma$ that includes only the individual mappings used in the evaluation of $[\![e]\!]^{\mu}_{\sigma}$.
- $\bar{\mu} \sqsubseteq \mu$ that includes only the individual mappings used in the evaluation of $[\![e]\!]^{\mu}_{\sigma}$.

For the general case, assume that we already have an interpolant Ψ for a state s' . The computation of $\mathsf{pre}(t, \Psi)$ can now be described as a relaxation of s by means of:

- ("Deletion") deleting any individual key-value pair in σ, μ, and any constraint in Π that Ψ does not depend on. This can be assisted by constructing a "dependency graph" during symbolic execution.
- ("Slackening") generalizing any equation of the form $v = e$ in Π, where v is a variable and e an expression, to become an inequality. That is, to become of the form $v \leq e'$ or $v \geq e'$ as appropriate – where e' is derived from e by changing some constants in e – such that the resulting state still implies the weakest precondition of t wrt. Ψ. Similarly for a key-value pair (v, e) in σ, μ, it can be replaced by a pair (v, fv) where fv is a fresh logical variable and a constraint $fv \leq e'$ or $fv \geq e'$ can be added to the path condition as appropriate.

B. Subsumption

Another key implementation feature concerns subsumption, as in Definition 5. More precisely, we wish to determine if one symbolic state s_2 implies another s_1 (this latter being an interpolant), as in (3). Our objective is to *transform* the entailment problem, which has existential quantifiers consequent $(\exists \tilde{I}_1 \tilde{A}_1)$ so that it can be effectively solved by a standard solver (in our case, Z3).

Implementing a perfect transformation is challenging. But in practice what we require is not a *complete* solution to the subsumption check, which is invoked at almost every step of symbolic execution, but an *opportunistic* approach using a cheaper (though incomplete) algorithm whose cost is not significantly more than *linear*. Toward this goal, TRACER-X implements the following. In the formula (3): replace each variable $a_1 \in \tilde{A}_1$ by a variable $a_2 \in \tilde{A}_2$ and replace each variable $i_1 \in \tilde{I}_1$ by a simple expression on some variables in $\tilde{I}_2$. Then we can *remove the existential quantification* on a_1 and i_1.

Essentially, this means to transform an entailment of the form $\forall i_2 \cdot F_2 \models \exists i_1 \cdot F_{12}$ into $F_2 \models F_{12}[i_2/i_1]$ where $F_{12}[i_2/i_1]$ denotes the result of substituting out i_1 with an expression of i_2 in F_{12}. This process is repeated until the consequent contains no more existential logical variables (symbolic inputs and symbolic addresses). The resulting entailment can then be solved by a quantifier-free solver.

Let us focus the discussion on symbolic addresses. The big (and remaining) question is, of course, how to determine, given a_1, which variable amongst $\tilde{A}_2$ is a_2? Our algorithm performs "matching" as follows:

- We make use of the equations $\forall x \in dom(\sigma_1) \cdot \sigma_1(x) = \sigma_2(x)$ and start with the program variables $x \in dom(\sigma_1)$ to perform each binding between $\sigma_1(x)$ and $\sigma_2(x)$ which will necessarily match some $a_1 \in \tilde{A}_1$ with some $a_2 \in \tilde{A}_2$.
- For those already matched $a_1 \equiv a_2$ we follow the *pointer chain*, making use of the equations $\forall a \in dom(\mu_1) \cdot \mu_1(a) = \mu_2(a)$ to further match $\mu_1(a_1)$ with $\mu_2(a_2)$. The process is repeated until: (a) a conflict is detected (subsumption check fails); (b) all the variables $a_1 \in \tilde{A}_1$ have been matched; or (c) a fixpoint has been reached.

This method is not complete, that is, it may not be able to eliminate all the logical variables $\tilde{I}_1$ and $\tilde{A}_1$ in the consequent.

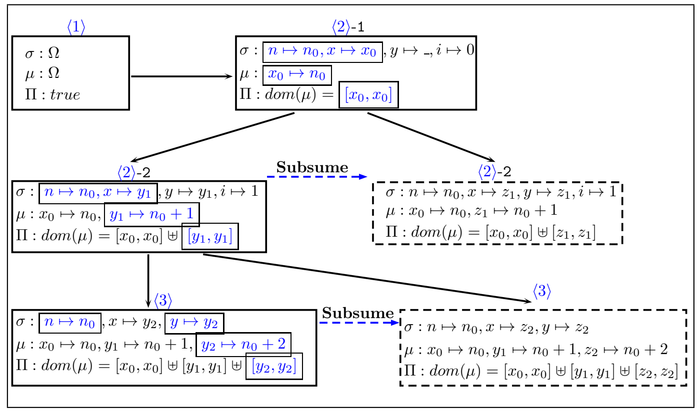

Fig. 3: Symbolic Execution Tree (SET) with Pruning

However, as we will show in our experimental evaluation, this method seems to be sufficient in practice.

C. Motivating Example Revisited

For presentation purposes, we use the reduced $MAX = 2$ and only show in Figure 3 an extract of the SET for relevant program points, namely $\langle 1 \rangle$, $\langle 2 \rangle$, and $\langle 3 \rangle$. For simplicity, we also perform *static unrolling* of the loop before building the SET[2], leading to two different versions of program point $\langle 2 \rangle$, namely $\langle 2 \rangle$`-1` and $\langle 2 \rangle$`-2` in the Figure.

For this example, the path condition only contains the constraints enforcing that the allocated memory regions are separated. Now consider the state at program point $\langle 3 \rangle$ in the leftmost path, right before the assertion `*y >= MAX + n`. The validity of the assertion is proved by showing that:

- $\sigma(y) \notin dom(\mu)$ contradicts the path condition. This implies that the dereference `*y` is memory safe. This proof leads to the fact that the learned interpolant must retain the constraint specifying that the domain of μ includes $[y_2, y_2]$ and the pair $(y \mapsto y_2)$ in σ.
- executing `assume(load(y) < 2 + n)` will result in an infeasible state. Note that the assertion will be compiled explicitly to `assume` and `error` transitions. This proof leads to the fact that the learned interpolant must *also* retain the pair $(n \mapsto n_0)$ in σ and $(y_2 \mapsto n_0 + 2)$ in μ.

The combined interpolant is presented by putting the retained mapping and constraint in boxes (and also in blue color).

Now consider the neighbor path reaching the same program point $\langle 3 \rangle$. At this state, `y` is holding the value of a symbolic address z_2. In order to perform the subsumption check, using the previously computed interpolant, we need to first eliminate the existential variable y_2. Note that the subsumption condition requires the current store $\sigma \equiv n \mapsto n_0, x \mapsto z_2, y \mapsto z_2$ must contain the pairs $(n \mapsto n_0)$ and $(y \mapsto y_2)$. This forces us to match y_2 with z_2. Subsequently, the subsumption check holds, thus we don't need to expand this state further.

Finally, we remark on how the interpolant at $\langle 2 \rangle$-2 is computed: by first simply retaining the pairs and constraints that the computation of the pairs and constraints in its children interpolants depends on. Additionally, we also need to retain

[2]Otherwise there will be infeasible paths caused by the loop counter i.

the information that guarantees the safety of transiting from the current state to its children. Specifically, $(x \mapsto y_1)$ and $[y_1, y_1]$ are kept in σ and Π respectively because variable `x` is dereferenced going from $\langle 2 \rangle$-`2` to $\langle 3 \rangle$.

VI. Experimental Results

We use a small collection of programs. Our first group of two are our academic motivating example in Figure 1, and the `regexp` (from KLEE tutorial 2 [18]) in Figure 2. The next group comprises three programs `basename`, `cut` and `pathchk` from the GNU Coreutils 6.10 benchmark. Finally, our last group comprises two substantial programs `statemate` and `nsichneu` from the Mälardalen WCET benchmark [19]. These latter two programs are specifically chosen because they exhibit many infeasible paths, and therefore represent a big challenge for symbolic execution. (In fact, existing specialized WCET algorithms struggle to analyze these programs precisely [14].) The target property for all runs is memory safety of pointer dereferences (which is automatic in KLEE). We use a DFS search strategy, justified because our experiments are to explore the entire search space.

For each program, we consider a small, medium and large versions of the underlying size parameter (typically an array bound), to demonstrate complexity. In our motivating example, these three values for `MAX` are 9, 18 and 27. For `regexp`, we use a fixed SIZE of 7, while varying the length of the constant string, displayed as ``` ``hello'' ``` in Figure 2, by *symbolic* strings of length 4, 5 and 6. Recall that in `regexp`, the number of paths is exponential in the bound. Therefore any non-pruning system will be limited to small problems.

We note that in the Coreutils programs, we were unable to run LLBMC on `basename`, because it erroneously reports a safety violation and terminates early. In the `cut` example, the breakdown into small or medium or large is not appropriate for LLBMC because these numbers pertain to the size of symbolic file, and this cannot be adjusted in LLBMC.

The results need little elaboration. One noteworthy point is in the small version of `pathchk`, KLEE is faster than TRACER-X, but then loses in larger versions of the problem. This is explained by the throughput (rate of LLVM instructions emulated) of KLEE, which is much faster than that of TRACER-X (since it does not compare paths). But as the experiments show, the exploration space is much larger for KLEE (across all programs and not just these Coreutils programs), and therefore pruning is eventually more effective.

Finally, on `nsichneu`, `statemate`, we do not consider various size instances because they use data structures of a fixed size.

We use Linux boxes with 16GB RAM and Intel Core i5 3.20 GHz for the Coreutils problems and i7 2.60 GHz for the others. In Table VI, the performance of KLEE and TRACER-X for each run is a pair: time in seconds (s), and number of instructions (i) executed. For LLBMC, we only report the running time. A one-hour timeout is indicated by ∞.

PROG.	TOOL	SMALL	MEDIUM	LARGE
`malloc` (Motivating example)	L	2.2_s	3004.7_s	∞
	K	0.2_s $1.1e4_i$	25.33_s $5.5e6_i$	∞
	T	0.03_s 403_i	0.03_s 817_i	0.05_s 1231_i
`regexp`	L	∞	∞	∞
	K	228.9_s $7.5e6_i$	∞	∞
	T	1.1_s $2.9e4_i$	2.9_s $6.3e4_i$	10.8_s $1.3e5_i$
`basename`	L	*error*	*error*	*error*
	K	383.9_s $3.8e8_i$	2797.4_s $2e9_i$	∞
	T	39.4_s $1.2e6_i$	40.4_s $1.2e6_i$	48.9_s $1.4e6_i$
`cut`	L	299.06_s		
	K	293.1_s $1.3e8_i$	489.7_s $2.9e8_i$	860.9_s $6.5e8_i$
	T	28.3_s $2.7e5_i$	24.3_s $1.7e5_i$	27.1_s $2.3e5_i$
`pathchk`	L	390_s	∞	∞
	K	139.11_s $1.5e8_i$	291.47_s $3.1e8_i$	2366.3_s $2.4e9_i$
	T	274.81_s $3.1e6_i$	433.51_s $3.8e6_i$	1268.7_s $7.3e6_i$
`statemate`	L	412.56_s		
	K	600.3_s $1.7e7_i$		
	T	0.1_s 4135_i		
`nsichneu`	L	25.99_s		
	K	2247.9_s $4.9e6_i$		
	T	11.4_s 3711_i		

TABLE III
Experimental Results. L=LLBMC, K=KLEE, T=TRACER-X

The main point of our small experimental evaluation is to show that there are significant programs for which Tracer-X demonstrates an advance in performance by a large margin.

Limitations and Future Work: First, we remark that there exist other Coreutils benchmarks that KLEE and TRACER-X can run but both timeout. In other words, none achieves search space exhaustion. We do not include those benchmarks because no conclusive interpretation can be drawn, essentially due to the lack of good metrics to quantify the "path coverage" of a pruning system with lazy annotations (TRACER-X) vs. a non-pruning system (KLEE). This is left as future work.

Second, "symbolic array indices" always pose a problem for symbolic execution of heap-manipulating programs. This topic is left out in the paper. While we simply inherit the solution of KLEE in forward symbolic execution, *symbolic indices*, in general, can make our subsumption check ineffective. This is because under the existence of symbolic indices, subsumption checking includes the potentially hard problem of checking graph homomorphism. This is also an interesting topic for future work.

VII. Further Related Work

Abstraction learning in symbolic execution has its origin in [8], and is also implemented in the TRACER system [11], [12]. TRACER implements two interpolation techniques: using unsatisfiability core and weakest precondition (termed *postconditioned* symbolic execution in [20]). Systems that use unsatisfiability core and weakest precondition respectively include Ultimate Automizer [21], and a KLEE modification reported in [20]. The use of unsatisfiability core results in an interpolant that is conjunctive for a given program point and therefore requires less performance penalty in handling. In contrast, weakest precondition might be more expensive to compute, yet logically is the weakest interpolant, hence its use may result in more subsumptions.

Abstraction learning is also popularly known as *lazy annotations* (LA) in [9], [10]. In [10] McMillan reported experiments on comparing abstraction learning with various other approaches, including *property-directed reachability* (*PDR*) and *bounded model checking* (*BMC*). He observed that PDR, as implemented in Z3 produced less effective learned annotations. On the other hand, BMC technology, e.g. [22], [7], [23], [24], employs as backend a SAT or SMT solver, hence it employs learning, however, its learning is *unstructured*, where a learned clause may come from the entire formula [10]. In contrast, learning in LA is structured, where an interpolant learnt is a set of facts describing a single program point.

Recently *Veritesting* [25] leveraged modern SMT solvers to enhance symbolic execution for bug finding. Basically, a program is partitioned into *difficult* and *easy* fragments: the former are explored in DSE mode (i.e., KLEE mode), while the latter are explored using SSE mode with some power of pruning (i.e., BMC mode). Though this paper and *veritesting* share the same motivation, the distinction is clear. First, our learning is *structured* and has *customizable* interpolation techniques. Second, we directly address the problem of pruning in DSE mode via the use of symbolic addresses. In contrast, there will be program fragments where Veritesting's performance will downgrade to naive DSE, e.g. our motivating examples. In summary, we believe that our proposed algorithm can also be used to enhance Veritesting.

Our approach is also slightly related to various *state merging* techniques in symbolic execution, in the sense that both state merging and abstraction learning terminates a symbolic execution path prematurely while ensuring precision. State merging encodes multiple symbolic paths using `ite` expressions (disjunctions) fed into the solver. The article [26] shows that state merging may result in significant degradation of performance, which hints that complete reliance on constraint solver for path exploration, as with the bounded model checkers (e.g., CBMC, LLBMC), may not always be the most efficient approach for symbolic execution. [16] proposes a symbolic execution that is based on KLEE, addressing two problems of state merging: degradation of performance due to solver having to deal with disjunctions, and inability to control search strategy (the strategy is dictated by the solver's implementation), respectively using heuristics.

References

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *26th PLDI*. ACM, 2005, pp. 213–223.

[2] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *10th ESEC/13th SIGSOFT FSE*. ACM, 2005, pp. 263–272.

[3] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th OSDI*. USENIX Association, 2008, pp. 209–224.

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Comm. ACM*, vol. 59, no. 6, pp. 93–100, 2016.

[5] L. de Moura and H. Rueß, "Lemmas on demand for satisfiability solvers," in *5th SAT*, 2002.

[6] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *10th TACAS*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[7] "LLBMC: introduction," Mar. 2012. [Online]. Available: http://llbmc.org/

[8] J. Jaffar, A. E. Santosa, and R. Voicu, "An interpolation method for CLP traversal," in *15th CP*, ser. LNCS, vol. 5732. Springer, 2009, pp. 454–469.

[9] K. L. McMillan, "Lazy annotation for program testing and verification," in *22nd CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 104–118.

[10] ——, "Lazy annotation revisited," in *26th CAV*, ser. LNCS, vol. 8559. Springer, 2014, pp. 243–259.

[11] J. Jaffar, J. A. Navas, and A. E. Santosa, "Unbounded symbolic execution for program verification," in *2nd RV*, ser. LNCS, vol. 7186. Springer, 2011, pp. 396–411.

[12] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *24th CAV*, ser. LNCS, vol. 7358. Springer, 2012, pp. 758–766.

[13] D. Chu and J. Jaffar, "A complete method for symmetry reduction in safety verification," in *24th CAV*, ser. LNCS, vol. 7358. Springer, 2012, pp. 616–633.

[14] D. Chu, J. Jaffar, and R. Maghareh, "Precise cache timing analysis via symbolic execution," in *2016 RTAS*. IEEE, 2016, pp. 293–304.

[15] J. Jaffar, V. Murali, and J. Navas, "Boosting concolic testing via interpolation," in *21st FSE*, 2013, pp. 133–143.

[16] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *33rd PLDI*. ACM, 2012, pp. 193–204.

[17] "Tracer-X KLEE symbolic virtual machine," 2017. [Online]. Available: https://github.com/tracer-x/klee/tree/experiment-201705

[18] "Testing a simple regular expression library," 2017. [Online]. Available: http://klee.github.io/tutorials/testing-regex/

[19] "Mälardalen WCET research group benchmarks." [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[20] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Postconditioned symbolic execution," in *8th ICSP*. IEEE, 2015, pp. 1–10.

[21] M. Heizmann, J. Christ, D. Dietsch, J. Hoenicke, M. Lindenmann, B. Musa, C. Schilling, S. Wissert, and A. Podelski, "Ultimate automizer with unsatisfiable cores - (competition contribution)," in *TACAS '14*, ser. LNCS, vol. 8413. Springer, 2014, pp. 418–420.

[22] L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, "Context-bounded model checking with ESBMC 1.17," 2012, contribution to SV-COMP at TACAS 2012.

[23] E. M. Clarke, D. Kroenig, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *11th TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.

[24] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: systematic test case generation for dynamic analysis and measurement," in *20th CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 209–213.

[25] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *ICSE*. New York, NY, USA: ACM, 2014, pp. 1083–1094.

[26] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *9th RV*, ser. LNCS, vol. 5779. Springer, 2009, pp. 76–92.