Strings Enhanced Symbolic Execution

Treating Strings as ADTs in a KLEE/Z3 framework

May 18, 2018

Background

- String intensive programs are abundant, and analyzing them naively in a symbolic execution framework is hard: string library code is analyzed as a bundle with application code, spawning a huge amount of irrelevent states.
- String SMT solvers allow a direct encoding of string library code to SMT formulae, thus reducing dramatically the number of explored states.
- Several string solvers exist: Hampi, Kaluza, Pisa, Stranger CVC4, S3, Z3str3, and they have been used (to some extent) to analyze SQL querries and programs in Java, JavaScript and PHP.
- str.KLEE enables symbolic execution of arbitrary C programs, which are (considerably) more widespread than previously mentioned PL, and have a more complicated semantics. It uses KLEE together with Z3, state-of-the-art SE and solver.

Implementation (High Level) Description

- Resolving occurrences of program string variables is not trivial in C, which is weakly typed, and allows taking the memory location of stored variables: msg$\rightarrow$contents = (void *) &buf;
- We use the term abstract buffers to denote the corresponding solver string variables. As the program modifies strings with writes, strcpy etc. abstrct buffers accumulate these changes by keeping consecutive versions for abstract buffers.
- Using a many sorted solver inevitably introduces sort conversion issues. Suppose that a bitvector is added to an integer: (i $<<$ 5) + strlen(msg) and it is sound to do this addition in both bitvector and integer domains. Which expression should be converted? we say: whichever is faster!

Accelerations List

- Context Aware Sorts
  - ((*s) == 'a') vs. ((*s) << 5)
- Tailored String Semantics
  - int strcmp(s1,s2) → bool strcmp(s1,s2)
  - char *strchr(s,c) → bool strchr(s,c)
- Solver Performance Driven Query Rewriting
  - str.indexof → str.contains
  - str.indexof → str.len
  - automatic deducing of query invariants
- Reducing Number of States with C to C translations:
  char *f(char *d,char *s) { while (*d++ = *s++); } return d;
  → strcpy(d,s); return d+strlen(src);
- Caching reads/writes.
- Reducing number of generated abtract buffer versions.
- Under approximation of programs paths. For example,
  Ignoring toUpper and adding relevant constraints.

Accelerations :: Context Aware Sorts

- Sort conversions are expansive, and in some cases they can be avoided altogether. Think of the following examples:

  ```
  if ((*s) == ' ') { s++; }
  if (((*s) << 3) < 100) { s++; }
  ```

  Since the returned value from (*s) has a string sort, then in order to shift it left, it needs to be converted to a bit vector. In constrast, if it is simply compared to the white-space character, then it needs not be converted at all.
- A simple pre processing of the program can easily identify locations where such conversions are not needed.

Accelerations :: Tailored String Semantics

- The semantics of string library functions often contains more details then actually needed by users.
- For example, whenever two strings are compared with strcmp(s1,s2), the returned value is either 0 when they are identical, or the ascii difference between the first place of change.
- Usually users just ask whether the result is 0 or not. This gives a chance for a sound optimization, since it enables us to use the native returned boolean sort from the solver
- Similarly, it is quite usual to use strchr to check the existence of a certain character in a string:
  if (strchr(s, ' ')) { return -1; }
  This enables a faster query.

Accelerations :: Query Rewriting

- Some queries happen often, like boundaries checks before each strcpy operation. The solver tries to figure out whether the first null termination of the source can be larger than the destination's size. As it so happens, sometimes the source buffer has no other constraints, and so clearly the null termination can be the very last byte. In these cases, replacing str.indexof with str.len is much faster.
- In the strchr case described before, str.indexof can be replaced with str.contains which is much faster.
- Some queries deal with very large ($>$ 1000 bytes long) strings. As the solver struggles with the huge size, it is sometimes possible to automatically prove an equivalent satisfiable query with much shorter sizes, and infer back the model from the small string to the large original string

Implementation :: Details

- Keep KLEE's memory model: Each allocation has a MemoryObject.
- Use *markString* calls to tag memory objects which we want to consider as strings.
- Writes to strings get translated to $version_{i+1} = prefixVersion_i + +newChar + +suffixVersion_i$
- For each read we introduce a new BitVectorVariable, that is return of the read. Then we make a string out of it and assert it is equal to the desired read location
- Allocations of symbolic sizes are currently conretize to a value between 10 and 128. (Easy to change this)
- Currently support *strcmp*, *str*(n)*cpy*, *strchr* and *strlen*.
- Calls to these intercepted in SpecialFunctionHandler, arguments resolved to MemoryObjects and through them to AbstractBuffers. Then appropriate constraints are added to the state.

Implementation :: Integration with KLEE

- KLEE is very bitvector orientated: expressions need witdh, and symbolic things can only be Arrays of bitvectors.
- Width is important to corectly interface with Extract and Concat expression and Casting.
- Arrays are important to correctly interface with the solver and test case generation
- Currently we set the width of all Strings/Ints to 8 (se they fit in1 memory cell) and special case on type (we started to type the Expression language)
- Arrays for Strings are represented as 0 length Arrays of bitvectors (seems to work well enough so far)