Strings Enhanced Symbolic Execution

Treating Strings as ADTs in a KLEE/Z3 framework

March 20, 2018

Introduction

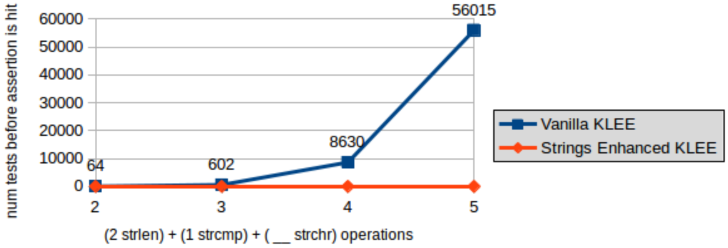
- While testing a program, SE engines like KLEE will usually test *actual implementations* of (strings) standard library functions (strcpy, strcmp, strchr etc.)
- Since strcpy, strcmp, strchr etc. often loop over the value of some *symbolic* variable, this results in state explosion.
- Ideally we would like to avoid forking states when they do not increase code coverage. A typical example is using strchr to check whether a character is present in a string or not. If the symbolic string is up to 100 characters long, then up to 100 states will be created, when in fact only 2 are needed.
- We offer to (cleverly) use a dedicated string solver in order to *lazily* evaluate the string operations.

Running Example

int main(int argc,char **argv)	>> With symbolic strings'
{	length bounded by 32,
if (strchr(p, '1')) {	each strchr operation
if (strchr(p, '2')) {	multiplies the existing
if (strchr(p, '3')) {	number of states by 32.
if (strchr(q, '4')) {	>> The same goes for strlen
if (strchr(q, '5')) {	and strcmp.
if (strlen(p) $\leq$ 6) {	>> By the time KLEE hits the
if (strlen(q) $\leq$ 6) {	null assertion, it has around
if (strcmp(p,q) = 0) {	2^{16} states ...
assert(0);	>> Using a dedicated string
}}}}}}}}}	solver reduces the number of
}	states to 10 ...

Table: State explosion with consecutive string library operations

Running Example



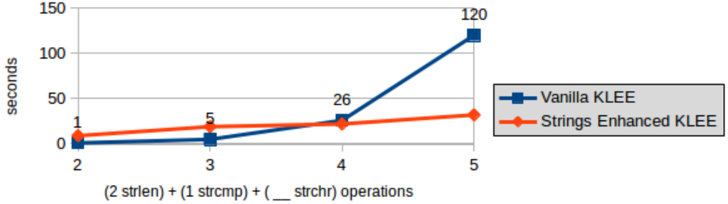


Figure: Consecutive string library functions cause state explosion.

Our Approach :: General Description

- When KLEE encounters a string allocation (on the stack, on the heap or as a constant string) it will **physically allocate** that requested memory. When KLEE updates a string p, during a write operation like **p[3]='M'**, it will do so with respect to the specific memory associated with p.
- Our approach uses the notion of **Abstract Buffers** (AB) and **versions**. We do not allocate physical memory for strings, but rather use string constraints to replace actual memory allocations and accumulate write changes.
- When **p = malloc(size)** is encountered, a new (Z3) string sort variable is introduced: $\mathrm{AB}_{serial=n,version=0}$, and a **length constraint** is added to the state: (= (str.len $\mathrm{AB}_{n,0}$) size).
- When **p[3] = 'M'** is encountered, an **additional version** is defined, and the write operation is expressed as string constraints between the last 2 versions.

Our Approach :: In Greater Detail :: Malloc

- KLEE's execution state adds these **data structures**:
  - static int numABs = 0;
  - std::map<std::string,int> var2AB;
  - std::map<std::string,ref<Expr>> var2ABoffset;
  - std::map<int,ref<Expr>> ABSize;
  - std::map<int,int> ABlastVersion;
- Effect of p = malloc(size) on **state's data structures**:
  - ++numABs;
  - var2AB[p] = numABs;
  - var2ABoffset[p] = ConstantExpr(0,Expr::Int32);
  - ABSize[numABs] = size; (size is a ref<Expr>)
  - ABlastVersion[numABs] = 0;
- Effect of p = malloc(size) on **state's constraints**:
  - int serialp = var2AB[p];
  - int versionp = ABlastVersion [serialp];
  - std::string name = makeName(serialp,versionp);
  - state.addConstraint(StrLengthExpr(StrVarExpr(name)) = size)

Our Approach :: In Greater Detail :: String Modifications

- Modifying a string with a write operation p[3]='M' or strcpy(p+offset,q) behave *very similar* in our approach.
- Effect of p[3] = 'M' on **state's data structures**:
  - ABlastVersion[var2AB[p]]++;
- Effect of p[i]='M' on **state's constraints**:
  - (= (str.len $AB_{n,d}$) (str.len $AB_{n,d+1}$))
  - (= (str.substr $AB_{n,d}$ 0 i-1) (str.substr $AB_{n,d+1}$ 0 i-1))
  - (= (str.at $AB_{n,d}$ i) (seq.unit c))
  - (= (str.substr $AB_{n,d}$ i+1 n-i) (str.substr $AB_{n,d+1}$ i+1 n-i))
- What if p[i]='M' writes outside allocated buffer? (next slide)

Our Approach :: In Greater Detail :: String Modifications

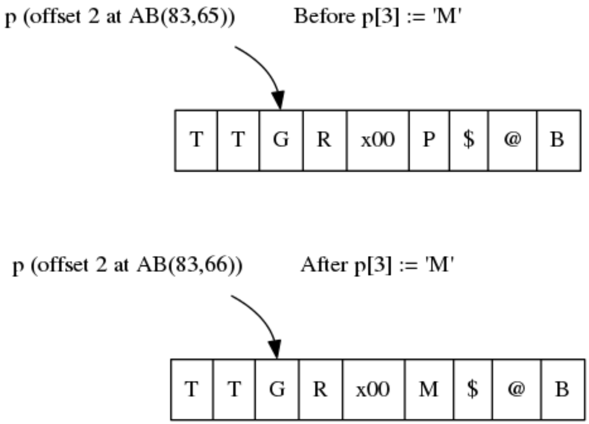


Figure: Writes are expressed as 4 string constraints between last and second to last versions of the abstract buffer: equal length, equal content before and after the write, and the modification itself.

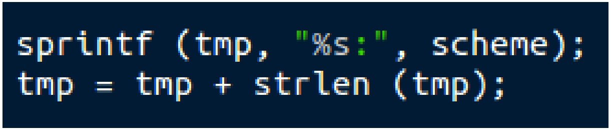
Our Approach :: In Greater Detail :: Access Violations

- The majority of buffer overflows occur in strings, so we obviously need to model that behaviour, and detect out of bounds reads and writes like vanilla KLEE does.
- Z3 semantics for out of bound string access does *not* always match our purpose, for example:
  (= 3 (str.len (str.substring s /*from*/6 /*length*/3)))
  is *not* true when s has length ≤ 5.
- We need to wrap every string action in an ITE Z3 expression(s), to make sure strings are not written (read) beyond their bounds. Note that these ITE's can range from simple ones for p[i] := 'M', to more complex ones needed to model strcmp(p,q), strchr(p,'M') etc.

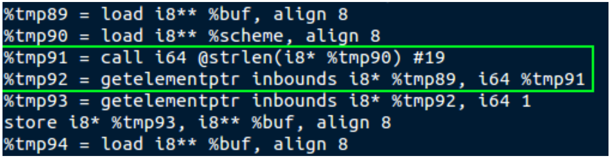
Our Approach :: Context Aware Z3 Sorts

- Currently KLEE only supports bit vectors, so adding a string sort is clearly the first step. Since string theory interfaces with integers rather than bit vectors, failing to add an integer sort support will surely result in bit blasting sort conversions.
- To minimize the need for sort conversions, we came up with the concept of **context aware Z3 sorts**.
- Recall the running example in [Vijay et al.] where strlen's result was an overflowing bit vector 16. And compare it to the example in the next slide where strlen can be safely handled as being an integer

Int Variable :: Example As Z3 Int Sort



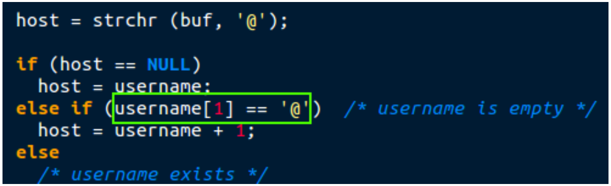
```
sprintf (tmp, "%s:", scheme);
tmp = tmp + strlen (tmp);
```



```
%tmp89 = load i8** %buf, align 8
%tmp90 = load i8** %scheme, align 8
%tmp91 = call i64 @strlen(i8* %tmp90) #19
%tmp92 = getelementptr inbounds i8* %tmp89, i64 %tmp91
%tmp93 = getelementptr inbounds i8* %tmp92, i64 1
store i8* %tmp93, i8** %buf, align 8
%tmp94 = load i8** %buf, align 8
```

Figure: int's can have integer sort whenever appropriate. In this example, using an integer as offset inside a buffer is safe.

Character Variable :: Example As Z3 String Sort


```
host = strchr (buf, '@');

if (host == NULL)
  host = username;
else if (username[1] == '@')  /* username is empty */
  host = username + 1;
else
  /* username exists */
```

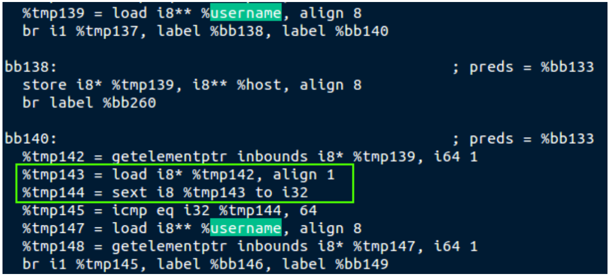
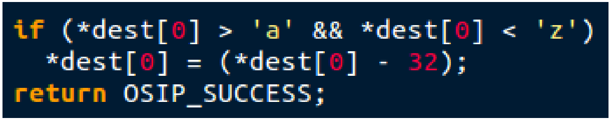
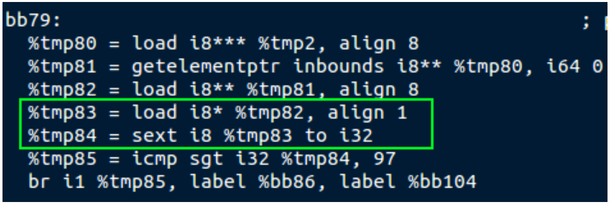

```
  %tmp139 = load i8** %username, align 8
  br i1 %tmp137, label %bb138, label %bb140

bb138:                                         ; preds = %bb133
  store i8* %tmp139, i8** %host, align 8
  br label %bb260

bb140:                                         ; preds = %bb133
  %tmp142 = getelementptr inbounds i8* %tmp139, i64 1
  %tmp143 = load i8* %tmp142, align 1
  %tmp144 = sext i8 %tmp143 to i32
  %tmp145 = icmp eq i32 %tmp144, 64
  %tmp147 = load i8** %username, align 8
  %tmp148 = getelementptr inbounds i8* %tmp147, i64 1
  br i1 %tmp145, label %bb146, label %bb149
```

Figure: Characters can have (length 1) string sort whenever appropriate. In this example tmp144 can be string compared to "@"

Character Variable :: Example As Z3 BitVec8 Sort



```
if (*dest[0] > 'a' && *dest[0] < 'z')
  *dest[0] = (*dest[0] - 32);
return OSIP_SUCCESS;
```



```
bb79:                                                        ;
  %tmp80 = load i8*** %tmp2, align 8
  %tmp81 = getelementptr inbounds i8** %tmp80, i64 0
  %tmp82 = load i8** %tmp81, align 8
  %tmp83 = load i8* %tmp82, align 1
  %tmp84 = sext i8 %tmp83 to i32
  %tmp85 = icmp sgt i32 %tmp84, 97
  br i1 %tmp85, label %bb86, label %bb104
```

Figure: Characters can have BitVec8 sort whenever appropriate. In this example tmp84 is sign compared to 'a'

Real Life Chalenges (oSIP2 CVE)

- A heap buffer overflow caused by a malformed SIP message of size 631 bytes with net payload size $= 51+72$ bytes. KLEE vanilla fails to find this bug after > 5 hours of running on server (though during that time it *did* find another less severe out of bound read access bug)
- Software packages often wrap library functions which require some find-and-replace human effort.
- Software packages sometimes re-write library functions with slight changes (libosip-strappend uses memmove, for example)