Lab: Code Safety: Types

Software, hardware, and systems should do everything that they are intended to do and nothing more. The “nothing more” part of that statement is what is meant by safety. If, say code, is allowed to do more than what is intended an attacker just might be able to exploit unintended execution of given code and perform some malicious task or compromise privacy, for example. Numerous programming languages require great care to develop a safe product. The C/C++ family is most notorious in allowing a developer to create exploitable code but other popular languages do so as well. Three classes of safety are of particular importance because they have been most responsible for unintended execution. These are: memory safety, thread safety, and type safety. This lab is concerned with type safety. By type safety we mean:

Type Safety:

1. No undesirable program behavior that is caused by a discrepancy between differing data types for constants, variables, and functions. Types could be manufactured (structs, classes) or primitive.
2. Integer overflow, underflow, casting should not cause undesirable program behavior.

Consider the following C++ code, provided in file `linear.c`, for illustration:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

// Input should be increasing set of distinct numbers
uint8_t inputOK (uint16_t elem[]) {
   int i;
   for (i=0 ; i < 232 ; i++) if (elem[i] >= elem[i+1]) return 0;
   return 1;
}

uint8_t lin_search (uint16_t key, uint16_t xs[]) {
   uint8_t i;
   if (inputOK(xs) == 0) return -1;
   for (i=0 ; i < 233 ; i++) if (key == xs[i]) return i;
   return -1;
}
```

The problem solved by function `lin_search` is: given a list (or sequence) `xs` of 233 positive distinct 16 bit integers in increasing order and a positive 16 bit integer `key`, find the position of `key` in `xs`. If `key` is not a member of `xs` then -1 is returned. For example, given `key` is 4 and `xs` is 1,2,3,4,5,6,7 then the position of `key` in `xs` is 3. Function `inputOK` is used to ensure integers in `xs` are distinct and increasing in order. If not, -1 is returned.

In file `linear_search.cry` Cryptol function `linSearchIdx` returns the position of a given `key` in given sequence `xs` or -1. Function `linSearchIdx` is simple: a sequence `s`, initialized with a single -1, is extended by comparing `key` with consecutive elements `x` of `xs` (via `x <- xs`). A counter `c` keeps track of the position in `xs` of element `x`. If `key` is not equal to `x`, then the next element in sequence `s` is the same as the previous element. If `key` has not yet been found in `xs`, that will be -1, but if key is equal to `x` the next element in `s` is the value of `c`. The purpose of `h` is to ensure that `s` contains the value of `c` when `key` equals `x` for all of `s` after equality is established (otherwise `s` is all -1s). The output of `linSearchIdx` is the value of the last element of `s` if `inputOK`, which has the same functionality as its namesake in the C code, returns `True` on `xs` or -1 if not. The signature of `linSearchIdx` is monomorphic: the `key` is 16 bits, each of 233 integers in `xs` are 16 bits and the position that is output is 8 bits. These numbers match those for the C code. Here is the Cryptol specification for `inputOK` and `linSearchIdx`:

```
// Returns True iff elmts is a sequence whose numbers are always increasing
// in order (all are distinct)
inputOK : [233][16] -> Bit
inputOK elmts = z ! 0
  where
    z = [True]#[ if x >= y then False else True /\ q
               | q <- z | x <- elmts | y <- (tail elmts) ]

// Returns the first location of a given key in the given list
// or -1 if the key is not in the list
linSearchIdx : [16] -> [233][16] -> [8]
linSearchIdx key xs = if (inputOK xs) then s!0 else -1
  where
    s = [-1]#[ if (key == x /\ h == -1) then c else h
             | x <- xs |  h <- s | c <- [0...]]
```

A SAW file `a.saw` is created to check the equivalence of `lin_search` and `linSearchIdx`. This requires `llvm` bitcode for functions in file `linear.c` which is obtained from this:

```
clang -g -O0 -emit -llvm -c linear.c -o linear.bc
```

File `a.saw` follows patterns established in the section on memory safety:

```
import "linear_search.cry";

let safe_setup = do {
   xs <- llvm_fresh_var "array" (llvm_array 233 (llvm_int 16));
   pxs <- llvm_alloc (llvm_array 233 (llvm_int 16));
   llvm_points_to pxs (llvm_term xs);
   key <- llvm_fresh_var "key" (llvm_int 16);
   llvm_execute_func [ llvm_term key, pxs ];
   llvm_return (llvm_term {{ linSearchIdx key xs }});
};

let main : TopLevel () = do {
   m <- llvm_load_module "linear.bc";
   saf_proof <- llvm_verify m "lin_search" [] true safe_setup z3;
   print "Done!";
};
```

Run `saw a.saw` to get this:

```
[13:35:52.130] Verifying lin_search ...
[13:35:52.140] Simulating lin_search ...
```

```
[13:35:59.688] Checking proof obligations lin_search ...
[13:36:00.684] Proof succeeded! lin_search
[13:36:00.684] Done!
```

The problem of finding the position of a key in an increasing sequence of distinct integers is normally solved using binary search. A common way this is presented is shown in file `bsearch-unsafe.c`:

```
// Input should be increasing set of distinct numbers
uint8_t inputOK (uint16_t elem[]) {
   int i;
   for (i=0 ; i < 232 ; i++) if (elem[i] >= elem[i+1]) return 0;
   return 1;
}
// Standard binary search except index can overflow for high enough f and l
uint8_t binsearch (uint8_t f, uint8_t l, uint16_t x, uint16_t elem[]) {
   uint8_t index = 0;
   while (f <= l) {
      index = (f+l)/2;
      if (x == elem[index]) return index;
      if (x < elem[index]) l = ((uint8_t)(f+l))/2-1;
      else f = ((uint8_t)(f+l))/2+1;
   }
   return -1;
}
// Call binary search, assume list size is 233
uint8_t b_search (uint16_t key, uint16_t xs[]) {
   if (inputOK(xs) == 1) return binsearch(0, 232, key, xs);
   else return -1;
}
```

For example, add the following `main` to the file:

```
int main (int argc, char **argv) {
   uint16_t elems[] =
      {  23,  26,  33,  40,  49,  50,  53,  60,  62,  67,  71,  77,  80,  85,
         94, 103, 109, 116, 117, 123, 124, 125, 126, 134, 135, 139, 140, 141,
        150, 152, 157, 165, 171, 173, 179, 184, 188, 197, 199, 206, 211, 218,
        222, 230, 232, 235, 243, 252, 253, 261, 265, 266, 267, 273, 281, 284,
        285, 286, 288, 289, 290, 299, 308, 316, 318, 323, 326, 333, 338, 344,
        349, 350, 353, 362, 371, 378, 380, 387, 393, 396, 402, 403, 407, 415,
        423, 426, 428, 436, 439, 445, 454, 459, 465, 473, 477, 486, 489, 497,
        503, 512, 516, 518, 527, 536, 537, 545, 551, 555, 561, 563, 570, 573,
        577, 579, 580, 582, 587, 591, 592, 601, 602, 603, 607, 614, 623, 632,
        638, 641, 648, 651, 653, 655, 660, 662, 663, 670, 671, 678, 679, 686,
        687, 695, 696, 700, 701, 703, 710, 717, 723, 732, 738, 744, 745, 746,
        750, 751, 752, 753, 757, 765, 770, 776, 777, 778, 787, 789, 797, 798,
        807, 808, 815, 816, 817, 824, 829, 830, 831, 834, 843, 851, 854, 859,
        864, 868, 874, 875, 879, 886, 889, 898, 903, 911, 918, 924, 925, 931,
        939, 948, 956, 964, 973, 980, 989, 998, 1004, 1008, 1009, 1017, 1025,
        1031, 1033, 1039, 1041, 1048, 1051, 1054, 1056, 1057, 1062, 1064, 1070,
        1075, 1084, 1086, 1091, 1098, 1099, 1103, 1108, 1117, 1121, 1124,
        1130 };
   printf("%d\n",b_search(atoi(argv[1]), elems));
}
```

Then run

```
[prompt]$ bsearch-unsafe 165
31
[prompt]$ bsearch-unsafe 587
116
[prompt]$ bsearch-unsafe 591
255
[prompt]$ bsearch-unsafe 1117
infinite loop
```

The last two errors are due to integer overflow in `(f+l)` when computing the midpoint. The following change to `binsearch`, in file `bsearch-safe.c`, uses a midpoint calculation that does not overflow:

```
uint8_t binsearch (uint8_t f, uint8_t l, uint16_t x, uint16_t elem[]) {
   uint8_t index = 0;
   while (f <= l) {
      index = (f+(l-f)/2);
      if (x == elem[index]) return index;
      if (x < elem[index]) l = (uint8_t)(f+(l-f)/2)-1;
      else f = (uint8_t)(f+(l-f)/2)+1;
   }
   return -1;
}
```

Compile bsearch-safe.c and run as follows:

```
[prompt]$ bsearch-unsafe 165
31
[prompt]$ bsearch-unsafe 587
116
[prompt]$ bsearch-unsafe 591
117
[prompt]$ bsearch-unsafe 1117
229
```

Next, use SAW to show `b_search` in `bsearch-unsafe.c` is not safe and `b_search` in `bsearch-safe.c` is equivalent to the search specification and is therefore safe. Run Cryptol and load `linear_search.cry`. Then run `:safe linSearchIdx` to show that `linSearchIdx` is safe. Create `llvm` bitcode for functions in `bsearch-unsafe.c` and `bsearch-safe.c` with this:

```
clang -g -O0 -emit-llvm -c bsearch-unsafe.c -o bsearch_unsafe.bc
clang -g -O0 -emit-llvm -c bsearch-safe.c -o bsearch_safe.bc
```

The SAW file for `bsearch-unsafe.c` is `b.saw` and looks like this:

```
import "linear_search.cry";

let safe_setup = do {
   xs <- llvm_fresh_var "array" (llvm_array 233 (llvm_int 16));
   pxs <- llvm_alloc (llvm_array 233 (llvm_int 16));
   llvm_points_to pxs (llvm_term xs);
   key <- llvm_fresh_var "key" (llvm_int 16);
   llvm_execute_func [ llvm_term key, pxs ];
   llvm_return (llvm_term {{ linSearchIdx key xs }});
};
```

```
let main : TopLevel () = do {
   m <- llvm_load_module "bsearch_unsafe.bc";
   saf_proof <- llvm_verify m "b_search" [] true safe_setup yices;
   print "Done!";
};
```

The SAW file for `bsearch-safe.c` is `c.saw` and looks like this:

```
import "linear_search.cry";

let safe_setup = do {
   xs <- llvm_fresh_var "array" (llvm_array 233 (llvm_int 16));
   pxs <- llvm_alloc (llvm_array 233 (llvm_int 16));
   llvm_points_to pxs (llvm_term xs);
   key <- llvm_fresh_var "key" (llvm_int 16);
   llvm_execute_func [ llvm_term key, pxs ];
   llvm_return (llvm_term {{ linSearchIdx key xs }});
};

let main : TopLevel () = do {
   m <- llvm_load_module "bsearch_safe.bc";
   saf_proof <- llvm_verify m "b_search" [] true safe_setup yices;
   print "Done!";
};
```