Assignment 1
Memory Safety

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Due Tuesday, October 8, 2024
125 points

In this lab you will implement a small imperative programming language and an analyzer to ensure memory safety. If you use one of the supported languages (OCaml, Python, or Rust) we provide you with starter code that contains a parser and some interface code to an external solvers of arithmetic constraints. You may appeal to us choose another common language, but you need to let us know by the end of Week 1 so we can make sure that the autograder can handle your implementation. For this lab you are encouraged to work in pairs. Each pair should hand in only one solution (after possibly multiple submissions), identifying the partner in Gradescope. You will hand in two solution files.

By Tue Oct 1: The file `lab1-tests.zip` that contains at least 10 test files for the interpreter and analyzer which should strike a balance of between safe and unsafe code. Your test files will be validated against a reference implementation, which means they should be syntactically correct and both evaluation on small inputs and safety analysis should take a reasonable amount of time. We will post regarding the availability of a reference implementation binary on the Andrew Linux machines so you can validate the test files yourself before you hand them in.

Use private Piazza posts to earn extra credit by pointing out bugs in the reference implementation of the interpreter and analyzer (we don't quite care the same way about the parser ...)

By Tue Oct 8: The file `lab1.zip` that contains your interpreter and analyzer. We will call

```
make
```

after unzipping this file. This should create two executables, `tiny_run` and `tiny_vc`. The first will execute sample programs, while the latter will verify their safety.

1 Example

Consider the file `example.tiny`.

```
requires 0 <= #input;

n := #input;

sum := 0;
i := 1;
while (i <= n)
  invariant 0 <= i /\ i <= n+1
  do
    sum := sum + i;
    i := i + 1
  done;

#output := sum
```

The `tiny_run` program should print one of the following and exit with the corresponding code from the table in subsection 2.3.

```
./tiny_run example.tiny 4 10      % 'success 10'
./tiny_run example.tiny xyz       % 'error'
./tiny_run example.tiny 4 11      % 'failure 10'
./tiny_run example.tiny -4 10     % 'abort'
```

The `tiny_vc` should verify the safety of this program and exit with a corresponding status code (as summarized in the table at the end of Section 3).

```
./tiny_vc example.tiny            % 'valid'
```

2 The Interpreter

The interpreter, `tiny_run`, should parse and execute the file as specified below. The starter code shows how to parse the given file, access the command line arguments, and exit with a particular status code.

The core of the interpreter has two tasks:

1. Verify that the program does not reference an undefined variable.
2. Execute the program and compare its output to the given output.

2.1 Undefined Variables

Determine algorithmically, as described in lecture, that all program variables are defined before their use. You should assume the variable `#input` is defined initially, and you should check that the variable `#output` is defined at the end. If not, `tiny_run` should print `undefined` and exit with status code 5.

We assume that all (in-bound) memory cells have been initialized to 0. In C, this would be the behavior of `calloc` and unlike `malloc` which does not guarantee initialization.

2.2 Evaluation

If the given program parses correctly and passes the def/use test, `tiny_run` should execute the program after setting the variable `#input` to the first command line argument and testing the precondition given by `#requires <form>`.

If the precondition is false, the interpreter should print `abort` and exit safely with status code 3. The same is true if a command `test <form>` fails.

Then the interpreter should execute the program as shown in lecture, following the definition of the language semantics. If it finishes normally, the value of the variable `#output` should be compared with the second command line argument. If they are equal, it should print `success` and the output value and exit with status code 0. If they are not equal, it should print `failure` followed the value of the output and exit with status code 2.

During evaluation, when there is an out-of-bounds memory reference, the program should print ‘unsafe‘ and exit with status code 4. The memory bound U is set to 1024, so any valid memory address i must satsify $0 \leq i < 1024$. In-bounds references that have not been initialized should return the value 0.

During evaluation, the interpreter should ignore `assert <form>` commands and loop invariants, which are in the code for the purpose of verification only.

2.3 Summary

The exit status for the interpreter is one of the following

message	**exit code**	**condition**
`success` *#output*	0	output correct
`error`	1	file does not exist, or file or input does not parse
`failure` *#output*	2	output incorrect integer
`abort`	3	program safely aborted (precondition or test not satisfied)
`unsafe`	4	unsafe behavior (memory access out of bounds)
`undefined`	5	undefined variable

3 The Analyzer

The call to ’make’ should also create the executable `tiny_vc`. It is called with `tiny_vc <filename>.tiny`.

The analyzer has the task of generating a verification condition for safety by computing the weakest liberal precondition $\mathsf{wlp}\ \alpha\ \top$ where α is the given program and $\top$ is the standard post-condition for safety.

The analyzer should **not** track the contents of memory. For example, the code

```
M[0] := 1;
i := M[0];
M[i] := 0;
```

should be considered *unsafe* because we do not take memory contents into account. That is, when reaching the third line, nothing is known about the value of i (but it is known to be defined by the assignment in line 2).

The analyzer must pass the verification condition it constructed to Z3 in order to check the validity of $P \to \mathsf{wlp}\ \alpha\ \top$, where P is the precondition given with `requires` P at the beginning of the file. Z3 may return an indication of validity, a countermodel, or unknown (if it can neither prove nor refute the verification condition). We lump together the latter two outcomes as `unsafe`.

The analyzer should separate out all the white boxed formulas and check them separately. This means all loop invariants are checked, even if they might reside in some unreachable code, as indicated in the paragraph starting "Alternatively" on page L7.8 of the lecture notes.

Here are the specified outcomes:

message	**exit code**	**condition**
`valid`	0	the program is proved safe
`error`	1	file does not exist, or does not parse
`unsafe`	2	the program could not be proved safe

4 Language Reference

Below is a specification of the grammar of Tiny.

```
<idstart> := [a-z_]
<idchar> := [a-z_0-9]

<var> := <idstart> <idchar>*
       | '#input' | '#output'

<num> := [0-9]+

<exp> ::= <num>
        | <var>
        | <exp> '+' <exp>
        | <exp> '-' <exp>
        | <exp> '*' <exp>
        | '(' <exp> ')'

<form> ::= 'true'
         | 'false'
         | <exp> '<' <exp>
         | <exp> '<=' <exp>
         | <exp> '==' <exp>
         | <form> '/\' <form>
         | <form> '\/' <form>
         | <form> '->' <form>
         | '~' <form>
         | '(' <form> ')'

<prog> ::= <var> ':=' <exp>
         | <var> ':=' 'M' '[' <exp> ']'
         | 'M' '[' <exp> ']' ':=' <exp>
         | <prog> ';' <prog>
         | 'assert' <form>
         | 'test' <form>
         | 'if' <form> 'then' <prog> 'else' <prog> 'endif'
         | 'while' <form> 'invariant' <form> 'do' <prog> 'done'
```

```
<file> ::= 'requires' <form> ';' <prog>
```

Ambiguities are resolved as follows:

- Arithmetic operators are left associative, with precedence `*` > `+` `-` (where + and − have equal precedence).
  This means that `3 * 4 + 5 * 1 - 2` is parsed as `((3 * 4) + (5 * 1)) - 2`. You can get `-x` by writing `0-x`.
- Logical operators are right associative, with precedence `~` > `/\` > `\/` > `->`.
  This means `~ x < 5 \/ x <= 5 /\ ~ y < 2` is parsed as `(~(x < 5)) \/ ((x <= 5) /\ ~(y < 2))`.
- Sequential composition `<prog> ; <prog>` is right associative.
  This means `x := 1 ; x := x+1 ; y := x-1` is parsed as `x := 1` followed by `x := x+1 ; y := x-1`.