Lab 1
Memory Safety

Due Tuesday, February 24, 2026
125 points

In this lab you will implement a safety analyzer for a subset of the C0 programming language. 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 solver of arithmetic constraints (Z3). If you decide to use another language, you are responsible for ensuring that your Makefile is able to compile it without additional package installation on the Ubuntu 22.04 base images used by the autograder. For this lab you are encouraged to work in pairs. Each pair should hand in only one solution after possibly multiple submissions. Identify your partner in Gradescope and link with them on `safetychecker.net`. You will submit test cases on `safetychecker.net` and submit your analyzer on Gradescope.

By Tue Feb 17: Submit ten C0 test cases on `safetychecker.net` (log in with your Andrew ID). You should submit five test cases that the reference implementation reports as `valid` and five test cases that it reports as `unsafe`. Your test cases should be syntactically correct and the safety analysis should take a reasonable amount of time.

By Tue Feb 24: The file `lab1.zip` that contains your analyzer. We will call

```
make
```

after unzipping this file. This should create the executable `c0_vc`. This will verify the safety of C0 programs.

1 Example

Consider the file `example.c0`, which calculates the sum of an array.

```
int[] main(int argc, int[] in)
//@requires argc >= 0;
//@requires \length(in) == argc;
//@ensures \length(\result) == 1;
{
    int sum = 0;
    int i = 0;
    while (i < argc)
    //@loop_invariant 0 <= i && i <= argc;
    {
        sum = sum + in[i];
        i = i + 1;
    }
    int[] out = alloc_array(int, 1);
    out[0] = sum;
    return out;
}
```

There are several things to note about this program that illustrate the subset of C0 that you will use in this assignment.

- It consists of a single function `main`, which takes an integer `argc` and an array `in`, returning an array. Programs in the C0 subset must have **exactly one** function with this identical signature.
- The postcondition characterizes the length of the return array so that callers are able to use it safely. While programs may have additional postconditions, they are required to have one which characterizes the exact length of the return array either as a constant, or an arithmetic function of `argc` and `in`.
- When the return array variable is declared, it is initialized. Array variables in the subset must always be initialized on declaration.
- Integer and integer array variables are the only allowed types.
- There is exactly one `return`, and it is the last statement of `main`. There can be no statements after a return, and return statements cannot reside within conditionals, loops, or nested blocks.

Running `c0_vc` should verify the safety of this program and exit with a corresponding status code.

```
./c0_vc example.c0            % 'valid'
```

2 The Analyzer

The call to 'make' should also create the executable `c0_vc`. It is called with `c0_vc <filename>.c0`.

The analyzer has the task of generating verification conditions for safety by computing the weakest liberal precondition wlp α $\top$ where α is the given program command and $\top$ is the standard postcondition for safety. All integer constants and variables are modeled as 64-bit machine integers, and arrays range over 64-bit integer indices mapping to 64-bit integer values.

The analyzer must pass each verification condition it constructs to Z3 in order to check the validity of $P \to$ wlp α $\top$, where P is the precondition given by the `//@requires` clauses at the beginning of the file. Z3 may return an indication of validity, a countermodel, or unknown; we lump together the latter two outcomes as `unsafe`.

Your analyzer should track the contents of arrays using Z3's theory of arrays. Concretely, reads and writes to an array's contents are modeled using Z3's `Select` and `Store`. For example, the code

```
int[] a = alloc_array(int, 2);
a[0] = 1;
int x = a[0];
a[x] = 0;
```

should be proved *safe*: after the third line we know $x = 1$, and therefore the access `a[x]` is in bounds. On the Z3 side, the write `a[0] = 1` corresponds to updating the contents term to `Store(a_data, 0, 1)`, and the read `x = a[0]` corresponds to the constraint `x = Select(a_data, 0)`.

Your analyzer should separate out all the white boxed fuormulas and check them separately. This means all loop invariants are checked, even if they might reside in unreachable code.

The specified outcomes are shown below.

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

2.1 Memory Model

In our C0 subset, we support only 1D integer arrays (`int[]`). Multi-dimensional arrays, structs, and pointers (`int*`) are not supported. In addition, our subset of C0 imposes a **no-aliasing** restriction on arrays:

- The parser rejects any statement of the form `A = B;` where both `A` and `B` have type `int[]`.
- The parser also rejects any declaration initializer of the form `int[] A = B;` where `B` has type `int[]`.
- Assignments to indexed array expressions (e.g., `A[i] = e;`) are still allowed.

Because arrays cannot be aliased in this subset, you do **not** need to model a separate heap of reference IDs. You can model each `int[]` variable directly by its *length* and *contents*.
Concretely:

- `alloc_array(int, n)` initializes a fresh array value of length n whose elements are all 0.
- Reassigning an `int[]` variable via `alloc_array` replaces its array value.

2.2 Verification Goals

Your analyzer must ensure the following safety properties are always satisfied:

1. **Array Safety**: For any access `A[i]` (read or write), you must prove that it is in bounds.
2. **Division Safety**: For any operation `x / y` or `x % y`, you must prove that the denominator is not zero.
3. **Assertion Safety**: For any `assert(e)`, you must prove that it is safe.
4. **Return Value Length**: All programs must have a postcondition on `main` that characterizes the size of their return array value. This contract must be proved valid.
5. . **Error Statements**: Any `error(...)` statement should be treated as **always safe**.

3 Language Reference

We support a subset of C0 with basic support for arrays. Your input file must contain a single function, `main`, and it must be the only function in the file. It must occur immediately at the start of the file, ignoring leading whitespace and comments.

- Signature: `int[] main(int argc, int[] in)`
- Exactly one `return` statement is allowed.
- That `return` statement must be the final statement of `main`'s body, at top-level (not nested in any `if`, `while`, or block).

`main` returns an `int[]` value `\result`. Every program must include an `ensures` clause that specifies the length of the returned array exactly:

- `//@ensures \length(\result) == ...;`

There are no implicit postconditions about the return value beyond user-provided `ensures` clauses.

This subset does **not** support pointers, so it does **not** include the `NULL` constant. In particular, arrays cannot be compared to `NULL` (as in C), and any use of `NULL` is rejected by the parser.

Instead, if an `int[]` variable is declared without being initialized by `alloc_array`, we treat it as containing a default array value of length 0. Intuitively, this means that a check like "`A != NULL`" from pointer-based C corresponds to "$\mathsf{length}(A) > 0$" in this array-only subset.

3.1 Safe Expressions

In lecture, we assumed that all expressions are safe. Any operation that can result in safety violations, such as array accesses or division, must occur isolated in a single statement. The parser provided in the starter code automatically applies this transformation, hoisting any potentially unsafe operation out of expressions and into temporary variable assignments. So for example, the code:

```
if (A[i] > 10) { ... }
```

is automatically transformed by the parser into:

```
int _t0 = A[i];
if (_t0 > 10) { ... }
```

3.2 Grammar

The grammar below describes the **surface syntax** accepted by the parser.

```
<program> ::= 'int' '[]' 'main' '(' 'int' 'argc' ',' 'int' '[]' 'in' ')'
              <contract>* <block>

<contract> ::= '//@requires' <logic_exp> ';'
             | '//@ensures' <logic_exp> ';'

<type> ::= 'int' | 'bool' | 'int' '[]'

<exp> ::= <num>
        | <var>
        | 'true' | 'false'
        | <exp> '+' <exp> | <exp> '-' <exp>
        | <exp> '*' <exp> | <exp> '/' <exp> | <exp> '%' <exp>
        | '!' <exp>
        | <exp> '&&' <exp> | <exp> '||' <exp>
        | <exp> '==' <exp> | <exp> '!=' <exp>
        | <exp> '<' <exp>  | <exp> '<=' <exp> ...
        | <exp> '[' <exp> ']'
        | '(' <exp> ')'

<stmt> ::= <type> <var> [ '=' <exp> ] ';'
         | <type> <var> '=' 'alloc_array' '(' 'int' ',' <exp> ')' ';'
         | <exp> '=' <exp> ';'
         | '{' <stmt>* '}'
         | 'if' '(' <exp> ')' <stmt> [ 'else' <stmt> ]
         | 'while' '(' <exp> ')' <inv>* <stmt>
         | 'return' <exp> ';'
         | 'assert' '(' <exp> ')' ';'
         | 'error' '(' <string> ')' ';'

<inv> ::= '//@loop_invariant' <logic_exp> ';'

<logic_exp> ::= <exp>
              | '\length' '(' <exp> ')'
              | '\result'
              | '\forall' 'int' <var> '.' <logic_exp>
```

4 Using the reference implementation

Option 1: CLI The starter code includes a script `analyze.sh` that submits a test case to the course reference implementation hosted at `safetychecker.net`. The script first has you authenticate so that your submissions are attributed to your account. To authenticate, run `./analyze.sh --login` from the starter directory. This should open `safetychecker.net/cli` in your browser, where you **sign in with your Andrew ID** and copy a CLI token to paste back in the terminal. The token is cached locally so that subsequent runs of `./analyze.sh` can submit without asking you to log in again. Once you are logged in, you can submit a file by running `./analyze.sh -f your_test.c0`. The script prints the reference outcome, which is `valid` when the test is safe and `unsafe` when the test is unsafe. If you have trouble running the script, `./analyze.sh --doctor` reports whether the required command line tools are installed.

Option 2: Web Interface Alternatively, you can also submit tests directly on `safetychecker.net`. After signing in with your Andrew account, use the `Submit test` button in the sidebar to upload a C0 file. The sidebar shows your submitted files and whether each test is `valid` or `unsafe`. You can also view detailed information about the verification conditions generated by the reference implementation, and when they are available, counterexamples to failing verification conditions.

Partnering This lab supports working in pairs. In the user menu on `safetychecker.net`, use `Add partner` to enter your partner's Andrew ID or email address. If another student requests to pair with you, you will see an incoming request in the same user menu and can accept or reject it. Once you are linked, your submissions are shared between you and your partner, and both of you can view and delete the shared submissions. If you later un-link, new submissions are no longer shared.