Formal Program Verification

Rigorous Proof of Program Correctness and Security

Charles Averill

Computer Security Group
The University of Texas at Dallas

Saturday, 25 March, 2023

Who am I?

- 20 years old
- Undergraduate CS, Physics senior at University of Texas at Dallas
- Researching formal binary verification with Dr. Kevin Hamlen
- Studying quantum mechanics for my physics degree
- Officer of UTD Computer Security Group
- Applying to PL Ph.D. programs in Fall 2023
- Teaching a Practical Compiler Design course this semester: https://www.charles.systems/PCD
- My dog's name is Beth

Outline I

Outline II

What is "Verifying a Program"?

When we write code, we generally aim to induce some well-defined behavior in our computer. This behavior could be very simple:

$$f(x, y) = x + y,$$

or it could be very complex:

$$i\hbar \frac{d}{dt} \langle \Psi(t) | \Psi(t) \rangle = \hat{H} \langle \Psi(t) | \Psi(t) \rangle .$$

Regardless of the complexity of our programs, we typically want to ensure that they return the **correct answer**, and that they have no **insecurities**.

Program Correctness

We like our programs to be "correct". This means that they should return an answer in a form that we expect, no matter what (valid) input we give them.

The following addition routines in C are incorrect and correct, respectively, according to the standard definition of (modular) addition:

```
1  unsigned int add_incorrect(unsigned int x, unsigned int y) {
2      return x + y + 1;
3  }
4
5  unsigned int add_correct(unsigned int x, unsigned int y) {
6      return x + y;
7  }
```

$$\oplus(x, y, z) = (x + y) \bmod z$$

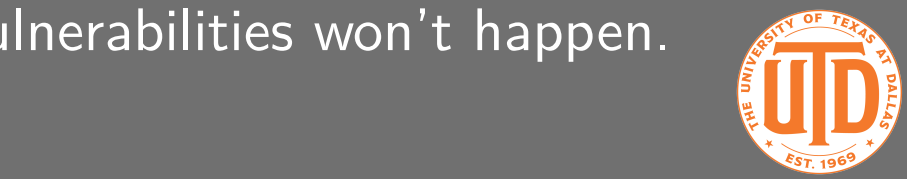
$$z = 2^{8 \cdot \text{sizeof(int)}} = 2^{32}$$

Program Security

We also like our programs to be "secure". This is a much more nebulous definition, but you will hear programmers claim that secure programs do not have "side effects". Side effects can arise from a large number of programmer errors, such as:

1. Out-of-Bounds Write (CWE-787)
2. Cross-Site Scripting (CWE-79)
3. SQL Injection (CWE-89)
4. ... There are ~600 more of these

The majority of these vulnerabilities exist solely due to poor input validation. A large part of program verification is ensuring that it properly handles input validation so that these vulnerabilities won't happen.



Program Security Examples

The following routines are insecure because they do not validate their inputs:

```
1  int add_pointer_contents(int *x, int *y) {
2      // What if x or y are a null pointer?
3      // Dereferencing those causes segmentation faults
4      return *x + *y;
5  }
6
7  int get_100th_element(int arr[]) {
8      // What if len(arr) < 100? We'll be accessing
9      // memory that we aren't supposed to
10     return arr[99];
11 }
```

Program Security Examples

Here are (more) secure variants:

```
1  int add_pointer_contents(int *x, int *y) {
2      // Still not very secure! What if somebody passes in
3      // garbage/freed pointers?
4      // There's really no easy way to tell! This is one reason
5      // why C is such an unsafe language
6      if (x == NULL || y == NULL) return 0;
7
8      return *x + *y;
9  }
10
11 int get_100th_element(int arr[], int default) {
12     if (sizeof(arr) / sizeof(arr[0]) <= 100) {
13         return default;
14     }
15
16     return arr[99];
17 }
```

Verification

Generally, correctness and security go hand in hand. If your code is not correct, it could return incorrect results to computations, propagating logical errors throughout your code. If your code is not secure, there is no guarantee that other activity on the host machine, regardless of intention, will not interfere with your routines.

Program "verification" is the process of ensuring that code is both correct and secure according to provided specifications of the intended behavior of the program, as well as fundamental rules of logic.

Proof Assistants

Verification can be achieved in many ways, but often is performed through the use of proof assistants: programming languages that use fundamental logic to confirm whether mathematical statements are true or not (more on this later).

```
space_data : list (nat * byte)
space_size, arr_size, arr_addr : nat
SPACE_DATA_IS_LEN_SPACE_SIZE : length space_data = space_size
SPACE_HAS_SIZE : 0 < space_size
ARR_HAS_LEN : 0 < arr_size
ARR_STARTS_WITHIN_BOUNDS : 0 <= arr_addr + arr_size < space_size
ARR_MAX_LEN : arr_size <= space_size - arr_addr
INDICES_START_AT_0 : fst (hd (0, "000"%byte) space_data) = 0
size0 : nat
Heqspace : Memory size0 [] = Memory space_size space_data
space0 : memory_space
a', size : nat
Heqarr : Array space0 (S a') size = Array (Memory size0 []) arr_addr arr_size
IHa' : Array space0 a' size = Array (Memory size0 []) arr_addr arr_size ->
       a' + fst (raw_mem_read (Memory size0 []) a') < a' + arr_size
```

```
(1/2)
S (a' + fst (raw_mem_read (Memory size0 []) (S a'))) < S (a' + arr_size)
```

```
(2/2)
S (a' + fst (raw_mem_read (Memory size0 (p :: data)) (S a'))) <
S (a' + arr_size)
```

How are Programming Languages Defined?

Depending on how much you've studied CS, you might have come across language "grammars", which describe what kinds of statements and expressions can exist within a program of a specific language. Consider the following grammar:

Syntax of the SIMPL Programming Language	
commands	$c ::= \texttt{skip} \mid c1;c2 \mid v := a \mid \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c2 \mid \texttt{while}\ b\ \texttt{do}\ c$
boolean expressions	$b ::= \texttt{true} \mid \texttt{false} \mid a_1\ \texttt{<=}\ a_2 \mid b_1\ \texttt{and}\ b_2 \mid b_1\ \texttt{or}\ b_2 \mid \texttt{!}\ b_1$
arithmetic expressions	$a ::= n \mid v \mid a_1\ \texttt{+}\ a_2 \mid a_1\ \texttt{-}\ a_2 \mid a_1\ \texttt{*}\ a_2$

Language grammars are nice, but they only tell us what a language **looks like**, not what it does.

Let's add this arithmetic expression to the grammar: $a_1 \models a_2$

We know what it looks like, but what is it supposed to do?

Semantics

This is where **semantics** steps in. A language semantics defines how certain statements in the language "act". There are a few ways to define language semantics:

1. Large-step operational semantics (what we'll look at)
2. Small-step operational semantics
3. Denotation semantics

Stores

The first fundamental concept to semantics is that of a **store**, or a relational mapping from variable names to variable values (in this case). Stores are intended to model a machine state (like a program's memory).

You can imagine stores like a dictionary in Python; they can be read from and assigned to:

```
1  store: Dict[str, int] = {}
2
3  store["sum"] = 0
4
5  for i in [5, 6, 7, 8, 9]:
6      store["sum"] += i
7
8  print(store["sum"])
```

Judgements

The second fundamental concept to semantics is that of a **judgement**. A judgement is a function that takes in an object from our grammar and returns either a new store or a value. These objects can be commands, expressions, etc.

```
1  def arith(expr, store) -> int:
2      match expr:
3          case SIMPL_CONST(n):
4              return n
5          case SIMPL_VAR(v):
6              return store[v]
7          case SIMPL_ADD(a1, a2):
8              return arith(a1, store) + arith(a2, store)
9          case SIMPL_SUB(a1, a2):
10             return arith(a1, store) - arith(a2, store)
11         case SIMPL_MUL(a1, a2):
12             return arith(a1, store) * arith(a2, store)
```

Judgements

We can write judgements another way: $\langle a_1 \texttt{ + } a_2, \sigma \rangle \Downarrow n$. We have similar judgements for the boolean expression values.

Command judgements converge to stores rather than values:

$$\langle \texttt{if } x \texttt{ <= } 3 \texttt{ then } x \texttt{ := } x \texttt{ + } 1 \texttt{ else skip}, \sigma \rangle \Downarrow \sigma'$$

We've defined how arithmetic and boolean expressions can be "evaluated" (they converge to a value), but we have not done the same for commands. To define the behavior of commands, we need **derivations**.

Derivations

Judgements are like mathematical propositions (e.g. "prime numbers have a maximum of 2 divisors" or "all odd numbers end in 5"), statements that are not necessarily true or false.

Derivations are proofs of those propositions. We represent proofs like this:

$$\frac{P \quad P \Rightarrow Q}{Q},$$

read as "If P is true, and P implies Q, then Q is true" (the law of Modus Ponens).

Here, Q is what we want to prove (the **goal**), so it goes on the bottom. P and $P \Rightarrow Q$ are "hypotheses", statements that must be true if the goal is to be true.

SIMPL Axioms

We need to define **axioms** (definitionally-true derivations) for SIMPL so that we can rigorously define how the language should operate.

Consider the following SIMPL axioms:

1. "`skip` does not modify the program store"

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma}$$

2. "Assignment statements update the program store with the value that a converges to"

$$\frac{\langle a, \sigma \rangle \Downarrow n}{\langle v \ := \ a, \sigma \rangle \Downarrow \sigma[v \mapsto n]}$$

There are more of these for the rest of the language, but we won't get into them.

Derivations

We can now use these axioms to determine the results of computations! Consider the following SIMPL program: $x := 5 * 25$. Let's look at its derivation using the assignment rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle a, \sigma \rangle \Downarrow n}}{\langle v := a, \sigma \rangle \Downarrow \sigma[v \mapsto n]}}{\langle a, \sigma \rangle \Downarrow n}}{\langle x := a, \sigma \rangle \Downarrow \sigma[x \mapsto n]}}{\dfrac{\dfrac{\dfrac{}{\langle 5 * 25, \sigma \rangle \Downarrow n}}{\langle x := 5 * 25, \sigma \rangle \Downarrow \sigma[x \mapsto n]}}{\langle 5 * 25, \sigma \rangle \Downarrow 125}}{\langle x := 5 * 25, \sigma \rangle \Downarrow \sigma[x \mapsto 125]}$$

The derivation has converged to a new σ, and we have that $\sigma(x) = 125$. This was a tiny program, but the same process allows us to compute the

So What?

We could already execute programs, this is nothing new. So why does it matter?

It matters because the act of executing a program is *identical to* the act of solving a mathematical proof by searching through a hypothesis space.

This provides us with access to thousands of years of study on mathematics and logic to formulate ideas about programs with! We can use these mathematical tools to prove all sorts of things about programs, notably their **correctness** and **security**.

What can we Prove about Programs?

Consider these two functions:

```
1  unsigned int double_int(unsigned int n) {
2      return 2 * n;
3  }
4
5  unsigned int mystery_function(unsigned int n) {
6      if (n != 0) return 2 + mystery_function(n - 1);
7      return 0;
8  }
```

What is special about the relationship between these two functions?

`mystery_function(0) = 0`, `mystery_function(3) = 6`, `mystery_function(7) = 14` ...

They look like they should perform the same function! But how do we test that? We could write unit tests... but there are a lot of unsigned integers! How can we know the outputs of these functions are the same for ALL inputs?

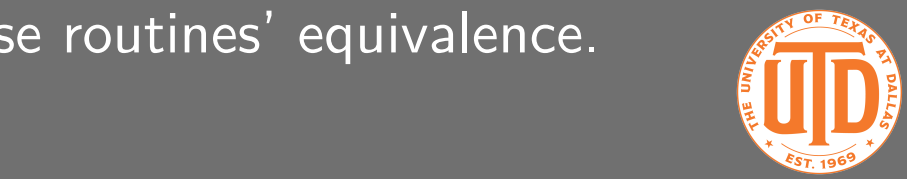
Coq Example

Coq is a proof assistant language. I'll convert our functions in question to its builtin functional programming language, Gallina. I get:

```
1  Definition double_simple (n : nat) : nat :=
2      2 * n.
3
4  Fixpoint double_recursive (n : nat) : nat :=
5      match n with
6      | O ⇒ 0
7      | S n' ⇒ 2 + double_recursive n'
8      end.
```

$(\mathrm{S}n' = n' + 1)$

Let's step through a simple proof of these routines' equivalence.



Double Routines Equivalence

```
1  Theorem double_simple_recursive_equivalence : forall (n : nat),
2      double_simple n = double_recursive n.
3  Proof.
4      intros. induction n as [| n' IHn' ].
5      - (* n = 0 *)
6          unfold double_simple. simpl. reflexivity.
7      - (* n = n' + 1 *)
8          simpl. rewrite ← IHn'.
9          unfold double_simple. rewrite ← mult_n_Sm.
10         assert (S (2 * n') = 2 * n' + 1).
11             rewrite Nat.add_1_r. reflexivity.
12         rewrite H. symmetry. trivial.
13 Qed.
```

Double Routines Equivalence

We've just proven that our recursive doubling algorithm always returns two times its input. We formulated this theorem by writing another algorithm to do the "simple" doubling, but we are not just limited to comparing the equality of program outputs given their inputs.

Using similar proof techniques, we can determine if a list-traversal algorithm ever touches certain items in the list, or how optimal a BST traversal algorithm is, or whether or not a function that accesses array memory ever tries to access out-of-bounds data.

The possibilities are (kind of) endless!

Proving a Program's Security

We've just proven the correctness of a program. But, it wasn't just any program, it was a program that was *written in a language used to prove things*.

That's a bit unfair! It also brings up the question: how is this method applicable to the real world? Is this just an academic curiosity?

It isn't! Let's get closer to proving something that seems like a program.

Functional -> Imperative

Our doubling routines were written in Gallina, a functional programming language.

FPLs abstract things like memory, clock cycles, etc. that are very familiar in the Von Neumann architecture (what all of our computers use). Therefore, to get closer to a "normal" imperative program, we will start by simulating a simple memory architecture.

Simulating a Memory Architecture

Code is available here if you'd like to follow along on your own machine:

https://gist.github.com/CharlesAverill/766c48a417aa7fec35168a26d1dc5546#file-formal_program_verification_examples-v-L44

General idea:

- We have a *Memory Space* that simulates all memory in a machine (think of RAM). Each place in the Memory Space is a pair (address, byte contents).
- We have *Arrays*. Arrays are attached to a Memory Space, have a starting address, and have a length.
- We have functions to read and write bytes directly from a MS. We also have some functions to read/write sequences of bytes from/to a MS that just use the raw IO functions multiple times.

The Theorem

Now comes the hard part. This is the theorem we want to prove:

```
1  in_arr_bounds
2      (fst (array_access_safe arr access_position)) arr.
```

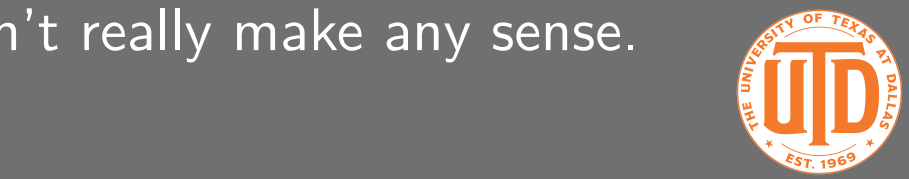
We've abstracted the idea of "array bounds" and "memory addresses" from mathematical concepts, so this is a mathematical proposition like any other. We want to prove that this statement is true.

There are some qualifiers we need to add to the theorem first, however.

Hypothesis Space

1. Memory space is non-empty (`0 < space_size`)
2. Array is non-empty (`0 < arr_size`)
3. Array starts within the bounds of the memory space
   (`0 <= arr_address + arr_size < space_size`)
4. Arrays cannot be longer than the distance between their address and the end of the memory space
   (`arr_size <= space_size - arr_addr`)
5. The indices of the memory space are well-ordered
   (start at 0 and strictly increase)

We assume all of these things to be true when we're writing the proof. We don't care about size-0 RAM, that doesn't really make any sense.



The Plan

To prove this statement, we need to use some induction. The general idea behind induction is that if I can prove a quality of numbers is true when:

1. The number is 0,
2. The number is some other number plus one,

then I have proved that the quality is true for all natural numbers ($n = 0, 1, 2, \dots$).

Execution

I won't be covering the actual proof, because it's pretty long. But we're going to prove the safety of our array access by an induction over the access position. So, if we prove that the array access is secure and correct when:

1. We're accessing the 0th element of the array,
2. We're accessing the $(n+1)$th element of the array, for an arbitrary n,

then we have proven that the array access is secure and correct for all of the quantifiers we listed in the hypothesis space: all memory spaces, all arrays in those memory spaces, and all access positions of those arrays.

The Challenges

Think of what we just (almost) proved: that one single byte read operation doesn't go out of bounds. We didn't even prove write security, much less the security of a more complicated operation, like computing the length of a string.

The current biggest challenge that comes with formal program verification is the difficulty of such proofs. We can add more preconditions to the hypothesis space, but in doing so we reduce the generality of our proofs, so they only verify specific instances of a program.

Another challenge is how to represent a more complex memory architecture, or even worse, a CPU with multiple instructions, pipelining, etc.

Complex Programs

The proof for single-byte reads was huge. Imagine the size of a proof for fibonacci number computations (a fairly simple algorithm that many people write at the beginning of their CS journeys).

Now consider trying to prove the amount of code reuse that would occur from trying to prove the correctness of a fibonacci x86 binary after proving the correctness of a fibonacci ARM binary.

It quickly becomes evident that we need some tool to write these proofs, to help minimize the workload of the verifier. Enter PICINÆ, "an infrastructure for machine-proving properties of raw native code programs without sources within the Coq program-proof co-development system."

PICINÆ

PICINÆ is being developed by the team led by my boss, Dr. Kevin Hamlen. Formal program verification has existed in various forms for decades, but primarily at the *source level* (before compilation). Unfortunately, many of the languages we use are built on top of C, an inherently unsafe language.

The solution? Verify programs at the binary level rather than the source level. All programs get executed as binaries in some form or another anyways, so going from the bottom-up is a more robust approach to formal verification.

I can't show any example proofs using the framework, but they're much longer than my attempt at the byte read operation. It's a work in progress though! Our goal is to make verification of binary programs much easier over time, so that it doesn't require a research team to verify software at the lowest level.

Questions?