CITS3007 Secure Coding
Formal methoids

Unit coordinator: Arran Stewart

Approaches to validating software

Suppose we have some function `int foo(int a, int b)`, and we want to ensure that for any `a` and `b` that are passed in:

```
foo(a, b) == foo(b, a)
```

How can we ensure that this is the case?

Approaches to validating software

In lectures and the project, one approach we've seen is *testing*.

We try to come up with representative values (often edge cases), and make sure those work.

e.g. we compare

foo(0,1)	vs	foo(1,0)
foo(-1,1)	vs	foo(1,-1)
foo(0,0)	vs	foo(0,0)

But this doesn't *guarantee* that the property holds.

What if we could prove a statement about *all* inputs?

e.g. "for all integers `a` and `b`, `foo(a,b) == foo(b,a)`"

Formal methods

If we are interested in ensuring some security property holds (e.g. a PNG file can be parsed safely, without causing buffer overruns) then again we can try to come up with edge cases.

(And tools like fuzzers might assist.)

But image manipulation routines are complex, and it's hard to be sure we've found all the edge cases.

Formal methods are a collection of ways we can get stronger guarantees that some property of a program holds.

Formal verification

The idea behind formal methods is that all programs are just very complicated mathematical functions. So we can do something like this:

1. Make a mathematical model of our software (ideally, automatically from our program)
2. Write down a property we want to hold – using some form of logic (e.g. predicate logic)
3. Prove that the model satisfies the property – either manually, automatically, or a combination

This is called formal *verification* – we *prove* the software has some property.

What kinds of properties do we care about?

For security, we often want to prove things like:

Memory safety	No buffer overflows, use-after-free, or null dereferences
Information flow	Secret data never reaches an untrusted output
Functional correctness	A parser accepts exactly the inputs it should
Access control	Unprivileged code can never escalate privileges

Formal verification spectrum

There are a spectrum of techniques we can make use of – from lightweight tools, to full mathematical proofs.

Broadly –

1. Testing – doesn't specify a property, doesn't provide a proof
2. Property-based testing – specifies a property, doesn't provide a proof
3. Model checking – specifies a property, automatically proves it
4. Interactive theorem provers – specifies a property, manually/interactively proves it

Property-based testing

This doesn't give us an actual proof – but it's a step beyond ad hoc testing.

The idea:

- You write down the property you want to check
- The computer generates test cases automatically

Property-based testing

In more detail –

1. User – defines some property P(x).
2. User – choose a strategy for generating values for "x"
3. Library – uses that strategy to generate random values for x, and tests whether the property holds
4. Library – if some counter-example is found, automatically shrink it to be as simple as possible, then fail the test.

Property-based testing

So instead of hand-picking test cases, we write down the *property* we want to hold, and let a tool generate the inputs automatically.

e.g. using Python's Hypothesis:

```
@given(integers(), integers())
def test_commutativity(a, b):
    assert foo(a, b) == foo(b, a)
```

- The tool runs thousands of randomly generated cases
- If it finds a counterexample, it *shrinks* it to the smallest failing input.

More useful properties

Useful properties to test include

- For parsers (like the one you have seen in the project) – for any structure X (e.g. an in-memory BUN file) –
  - When we serialise it then parse it, we get back the same structure
  - When given arbitrary input, the parser does not crash
- When we sort some array, the length remains unchanged
- When we compress then decompress some string S, we always get back the original string

Property-based testing in C

There are few property-based testing frameworks written specifically for C, though there are some (like theft).

For C++, RapidCheck is popular and well-maintained – and we can easily call C code from C++.

Simple RapidCheck Example

```
#include <rapidcheck.h>

bool is_sorted(const std::vector<int>& xs);

int main() {
  rc::check("sorting preserves size",
    [](std::vector<int> xs) {
      auto ys = xs;
      std::sort(ys.begin(), ys.end());

      RC_ASSERT(xs.size() == ys.size());
    });
}
```

Historical background

Property testing became widely known through **QuickCheck**, developed for the Haskell language.

The core idea was just:

> *Instead of writing many examples, describe what must always be true.*

QuickCheck influenced many later systems:

- RapidCheck (C++)
- Hypothesis (Python)
- proptest (Rust)
- jqwik (Java)

Csmith and compiler testing

One historically important project: Csmith

It

- automatically generates random C programs
- compiles them with many compilers
- compares behaviour

Compilers should agree on valid C semantics, so disagreements often reveal compiler bugs.

Csmith and compiler testing

Csmith found hundreds of bugs in GCC, LLVM, and commercial compilers.

It had to ensure generated programs avoided *undefined behaviour* (since if UB occurs, compilers can validly disagree on what to do).

This became a landmark demonstration of automated testing via generated inputs.

Property-based testing vs fuzzing

- libFuzzer
- AFL++
- Honggfuzz

Both are kinds of randomized testing, and both involve generated inputs.

Property testing tends to focus on semantic invariants and consistency properties.

Fuzzing primarily searches for crashes, hangs, and memory corruption.

So we can see it as a kind of property-based testing - it asks, “Is there a random input that can break the program?”

Property-based testing vs fuzzing

The boundary is blurry.

Modern fuzzers can incorporate:

- structure-aware generation
- grammar-based mutation
- coverage guidance
- invariant checking

Combining sanitizers with property-based testing

A good approach is to combine sanitizers with property-based testing.

Compile with e.g.

```
CFLAGS="-fsanitize=address,undefined"
```

Example: compression property

Suppose we have:

```
char *compress(const char *str);
char *decompress(const char *str);
```

Then a useful property is (in pseudocode):

```
decompress(compress(x)) == x
```

Model-checking

The key idea:

- You write down the property
- The computer proves it automatically

(Used in languages like F* and Dafny.)

Model-checking

- There used to be many different domain-specific model checkers.
- But it was later found that a set of tools called SMT solvers could solve problems from almost all domains
  - (SMT = "Satisfiability Modulo Theories")

Z3 is an example SMT solver.

Most solvers accept queries in a standard format called SMTLIB.

How model-checking works

1. User: writes down their property in a high-level language
2. Tool: compiles the property to a set of SMTLIB queries
3. Tool: calls an SMT solver to answer the queries
4. Tool: if counter-examples are found - converts them back into a form understandable by the user

Applications of model-checking

Use by:

- AMD, for checking correctness of computer-chip design
- Airbus – correctness of aeroplane control systems

Model-checking – security applications

We can use this approach to verify properties like

- "An attacker cannot learn the session key unless they already know the long-term secret"
- "No user can reach the admin state without passing the authentication state first"
- "The system never reaches a state where two nodes believe they are both leaders and deadlock" (denial of service)
- "You cannot call free() twice on the same pointer in any execution path"

Model-checking works well when there are *bounds* on the problem, and we can model our software as a finite-state transition system.

Bounded model checking

A key application of SMT solving is **bounded model checking** (BMC):

1. "Unroll" the program for up to k steps
2. Encode the unrolled program as an SMT formula
3. Ask: "can this assertion fail within k steps?"

Tools like CBMC (for C) work this way.

They can automatically verify absence of buffer overflows, integer overflows, and assertion violations *without* test cases –

But: only up to a fixed bound. Proving full correctness for *all* inputs, for *any* number of loop iterations, requires more.

Model-checking limitations

Model checking struggles with unbounded problems (e.g. where the heap can potentially grow without bounds, or a compression algorithm must work regardless of input size).

Model checking also doesn't immediately tell us *why* our program has gone wrong.

Interactive Theorem Provers

For complex properties, we may use **Interactive Theorem Provers**.

The idea:

- You write the proof
- The computer checks it

Interactive Theorem Provers

Unfortunately, common programming languages (Python, C, Java) are not very well designed for writing proofs in.

So we use custom language – theorem provers – in which proofs are easier to write. Examples: Rocq, Isabelle/HOL, Lean

Often, these theorem provers will *generate* code in C, Haskell, etc which satisfies the properties we want.

Theorem prover pros and cons

Pros:

1. We get a formal guarantee of correctness.
2. We can represent pretty much any proof or argument.

Cons:

1. Writing down a correct proof is typically many, many times more time-consuming than writing the program
2. We have to write our program in specialised languages

Takeaways

Property-based testing

- Quick to setup and run
- No reason not to use it!
- Much more powerful than traditional tests

Model checking

- Can be very powerful
- Tricky to use with complicated properties

Interactive theorem provers

- Time consuming to use
- But if human lives or hundreds of millions of dollars depend on code running correctly, we can justify the cost