Formal proofs of software, some perspectives

Jean-François MONIN

jean-francois.monin@imag.fr

Université de Grenobles Alpes, Verimag & LIAMA

Hanoi, October 17, 2016

Question

What do

- Bitcoin transaction scripting
- network packet filtering
- power management

have in common ?

Why formal proofs of software?

Objectives

- Bug-free critical components of software systems
- Complexity more and more challenging
- Formal proof technology can be applied
  - directly on components
  - or (additionally) on auxiliary tools: compilers,...

Application fields

Transportation, vehicles, aircrafts, powerplants, banking, telecom,...

Why bugs?

Systems

Result of design and implementation decisions
For actions requiring effort, decisions take time
(e.g. carrying heavy bricks, stones)

Software systems

Copy is for free
Result of many many design and implementation decisions
Most decisions take almost no time

Why bugs?

Conjecture 1

Comparing 2 systems built using the same amount of work time,
the software contains orders of magnitude more decisions
than the other.

Remark

Each decision is an opportunity of mistake

Corollary

Comparing 2 systems built using the same amount of work time,
the software contains orders of magnitude more mistakes
than the other.

Using software components

Analysing software components?

Complicated objects

No time to analyse them

Common receipe

Repeat until it works

- guess, make conjectures
- experiment

Makes the situation even worse!
Multiplication of approximately understood,
possibly unsuitable or buggy pieces of code

A piece of software

- can be seen as a gigantic formula
- written in some programming language
- itself designed using many design decisions

Some of them are wrong

E.g. : misleading use of good mathematical notations with another meaning

```
a = b + 1
i = i + 1
```

Hence $0 = 1$?

Conjecture 2

Writing good programs with badly designed langages is as easy as making calculations in the roman numeral system.

Another overlooked notion: sums of types

Data structures

- arrays, records
- lists, trees: pointers

Set theory

Cartesian products, unions, intersections (?)

Better: use type theory, related to proof theory

- products $\quad \times \quad \wedge$
- sums (disjoint unions), said otherwise choice $\quad \oplus \quad \vee$

Functional programming

- products $(a, b) = \lambda k.\, k\ a\ b$ with type $(A \to B \to X) \to X$
- sums or choice
  $inj_1\ a = \lambda k_1 k_2.\, k_1\ a$ with type $(A \to X) \to (B \to X) \to X$

Formal methods

Anyway, even with badly designed programming languages, it is possible to provide a mathematical definition of the meaning of a program.

Then it is possible to state logical conjectures on programs and to (dis)prove them.

Scientific background

Formal Semantics of programming languages

- Rule-based, providing a clear mathematical definition
  Natural semantics, Structural Operational Semantics

Secure proof assistants

- Higher-order logic; powerful type systems, inductive types
- Prominent instances: Isabelle, Coq

Well-defined programming languages

- Functional languages : Ocaml, Haskell...
  based on λ-calculus
- Dedicated languages, e.g.
  - Lustre
  - k-framework based on rewriting theory

Different approaches and tools to Formal Methods

- Hoare logic, Calculus of Weakest Preconditions
  imperative program = state transformer (forwards)
  = formula transformer
- B: refine set-theoretic imperative specifications into low-level programs
- Model checking: for concurrent systems
  compare temporal logic specifications with implementations
  extensions to real-time systems, hybrid systems
- Static analysis
  automated computation of soundness properties,
  e.g. about pointers and or array bounds
- Interactive proof assistants
  provide full power of maths

Coq, a secure proof assistant

Support to any mathematical activity

- Write definitions
- State and prove theorems

Applications in pure maths

- 4 colour theorem, odd-order theorem (finite group theory)
- category theory, higher-order homotopy theory

Applications in Computer Science

- Compcert : certified C compiler
- Verasco : certified static analyser
- Security API
- Distributed algorithms
- Many many others

LIAMA project FCST

Focuses on the correctness of auxiliary tools: compilers,...

- Certified compiler for Lustre
- DSL for OS kernels

Answer

What do

- Bitcoin transaction scripting
- network packet filtering
- power management

have in common ?

They all make use of in-kernel interpreters!

In-kernel interpreters

- In-kernel interpreters have become a staple of modern computation processes
- They also have become a major concern regarding security
- Risks of malicious attacks or intern errors
- In-kernel interpreters run in kernel space
- Any error or attack can have tremendous consequences

The BPF language

Originally serves for defining packets filters
Is a low-level language rather close to assembly
Is used here as a system call filter

Example of BPF

```
; load syscall number
ld [0]
; deny open() with errno = EACCES
jeq #SYS_open, L1, L2
L1: ret #RET_ERRNO|#EACCES
; allow getpid()
L2: jeq #SYS_getpid, L3, L4
L3: ret #RET_ALLOW
; allow gettimeofday()
L4: jeq #SYS_gettimeofday, L5, L6
L5: ret #RET_ALLOW
L6: ...
; default: kill current process
ret #RET_KILL
```

Each system call gets an entry in the list of rules, along with the expected behavior regarding this particular sytem call.

SCPL

A domain specific language for defining sytem call policies

- in a more user-friendly way
- less error prone
- reduces the risk of having incorrect BPF policies
- to be translated to BPF

Example of SCPL

```
{ default_action = Kill;
rules = [
{ action = Errno EACCES; syscall = SYS_open };
{ action = Allow; syscall = SYS_getpid };
{ action = Allow; syscall = SYS_gettimeofday };
...
] }
```

Thanks

Questions?