QuickChick

Speeding up Formal Proofs with Property-Based Testing

Cătălin Hrițcu

INRIA Paris-Rocquencourt

(Prosecco team, Place d'Italie office)

About me and how I use Coq

- **Working on formal methods for security, broadly**
- **Still rather naïve Coq user (after ~4 years of learning)**
- **Some teaching: Software Foundations and a bit of CPDT**
- **"Mechanized Metatheory for the Masses"**

Soundness of static & dynamic enforcement mechanisms

- expressive type systems using SMT solvers:
  ZKTypes [CCS 2008], F5 [TOSCA 2011, JCS2014],
  DMinor [ICFP 2010, JFP 2013], recently joined F* effort
- verification condition generator: DVerify [CPP 2011]
- certified translations: Expi2Java [NFM 2012]

  machine with dynamic IFC [S&P 2013, POPL 2014]
- micro-policies: generic hardware-accelerated tagging schemes

About me and how I use Coq

- Working on formal methods for security, broadly
- Still rather naïve Coq user (after ~4 years of learning)
- Some teaching: Software Foundations and a bit of CPDT
- "Mechanized Metatheory for the Masses"

- Devising correct security mechanisms is hard
  - full confidence only with mechanized proofs
  - this is why I'm a Coq addict

machine with dynamic IFC [S&P 2013, POPL 2014]

– micro-policies: generic hardware-accelerated tagging schemes

Problem: proving is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time
- **Proving does aid design ... but only at a very high cost**
  - most enlightenment comes from failed, not from successful proofs
  - a failed proof attempt is a very costly way to discover a design flaw
  - fixing flaws not always easy, might require serious redesign
  - failed proof attempt will generally not convince an engineer
  - proving while designing is frustrating, tedious, time consuming
- **Even when design correct & stable, proving still costly**
  - countless iterations for discovering lemmas and invariants
  - my proofs are often "fragile", so the cost of each iteration is high

Problem: proving is very costly

- **My proofs are boring, but designing security mechanisms is not**
  - definitions and properties often broken, and evolve over time
- **Proving does aid design ... but only at a very high cost**
  - most enlightenment comes from failed, not from successful proofs

- This is the itch I'm trying to scratch
  - other people might have similar itches though

- **Even when design correct & stable, proving still costly**
  - countless iterations for discovering lemmas and invariants
  - my proofs are often "fragile", so the cost of each iteration is high

Could **testing** help with this problem?

- Can property-based testing
  - lower the cost of formal proofs?
  - become an important part of the theorem proving process in Coq?
- Yes, I believe / hope so
  - own recent positive experience with testing
  - I'm not the only one (e.g. Isabelle, FocalTest, ...)

- We are basically just starting on this
  - A lot of research & engineering work left

This talk

- Introduction to property-based testing with QuickCheck
- Testing noninterference, quickly
- Polarized mutation testing
- A simple QuickCheck clone for Coq (prototype)
- Some ideas for deeper integration with Coq/SSReflect

Collaborators

Arthur Azevedo de Amorim
(UPenn, now INRIA intern)

Maxime Dénès
(UPenn)

John Hughes
(Chalmers)

Leo Lampropoulos
(UPenn)

Zoe Paraskevopoulou
(NTU Athens,
soon INRIA intern)

Benjamin Pierce
(UPenn)

Antal Spector-Zabusky
(UPenn)

Dimitris Vytiniotis
(MSR Cambridge)

An introduction to

PROPERTY-BASED TESTING WITH QUICKCHECK

[Claessen & Hughes, ICFP 2000]

```haskell
import Test.QuickCheck
import QuickCheckWithDiscards

f :: Int -> Int
f n = g 0 n
  where g a b
          | a==b        = a
          | c*c > n     = g a (c-1)
          | otherwise   = g c b
          where c = (a+b+1) `div` 2

prop_int_sqrt x =
  x >= 0  ==>  (f x * f x <= x  &&  (f x + 1) * (f x + 1) > x)
```

```
*Main> quickCheck prop_int_sqrt
+++ OK, passed 100 tests.
*Main>
*Main> quickCheckWithDiscards (stdArgs {maxSuccess = 1000}) prop_int_sqrt
+++ OK, passed 1000 tests.
Success {numTests = 1000, labels = [], output = "+++ OK, passed 1000 tests.\n"}
Actual tests run: 1000
Discards: 1014
Discard ratio: 0.503
```

```haskell
import Test.QuickCheck
import QuickCheckWithDiscards

f :: Int -> Int
f n = g 0 n
  where g a b
          | a==b        = a
          | c*c > n     = g a (c-1)
          | otherwise   = g c b
          where c = (a+b+1) `div` 2

prop_int_sqrt x =
  x >= 0  ==> (f x * f x <= x  &&  (f x + 1) * (f x + 1) > x)

prop_int_sqrt_small =
  forAll (choose (0, 10000)) prop_int_sqrt
```

Custom input data generator

```
*Main> quickCheckWithDiscards (stdArgs {maxSuccess = 1000}) prop_int_sqrt_small
+++ OK, passed 1000 tests.
Success {numTests = 1000, labels = [], output = "+++ OK, passed 1000 tests.\n"}
Actual tests run: 1000
Discards: 0
Discard ratio: 0.000
```

```haskell
import Test.QuickCheck

f :: Int -> Int
f 0 = 0
f 1 = 1
f x = 6
```

Broken definition

```haskell
prop_int_sqrt x =
  x >=0  ==>  (f x * f x <= x  &&  (f x + 1) * (f x + 1) > x)

prop_int_sqrt_small =
  forAll (choose (0, 10000)) prop_int_sqrt
```

```
*Main> quickCheck prop_int_sqrt_small
*** Failed! Falsifiable (after 1 test):
8709
*Main> quickCheck prop_int_sqrt_small
*** Failed! Falsifiable (after 1 test):
2036
```

Counterexample

```haskell
import Test.QuickCheck

f :: Int -> Int
f 0 = 0
f 1 = 1
f x = 6

prop_int_sqrt x =
  x >=0  ==>  (f x * f x <= x  &&  (f x + 1) * (f x + 1) > x)

prop_int_sqrt_small =
  forAll (choose (0, 10000)) prop_int_sqrt

prop_int_sqrt_small_shrink =
  forAllShrink (choose (0, 10000)) shrink prop_int_sqrt
```

```
*Main> quickCheck prop_int_sqrt_small_shrink
*** Failed! Falsifiable (after 1 test and 11 shrinks):
2
*Main> quickCheck prop_int_sqrt_small_shrink
*** Failed! Falsifiable (after 1 test and 10 shrinks):
2
```

Small counterexample

Own experience with

TESTING NONINTERFERENCE, QUICKLY

Can we quickcheck **noninterference**?

[ICFP 2013 and beyond]

- **Context**
  - designing **real machine with dynamic IFC** (>100 instructions)
- **Experiment**
  - **very simple stack machine** (10 instructions)
  - standard **end-to-end noninterference property**
  - manually **introduced 14 plausible IFC errors**, and **measured how fast they are found**
- **Encouraging results**
  - however, **not using QuickCheck naïvely**

3 secret ingredients

1. **Fancy program generation strategies**
   - $s_1 \approx s_2$ – generate s_1 then vary secrets to get $s_2 \approx s_1$
   - distributions, instruction sequences, smart integers
   - best one: “generation by execution”
     - **19 instructions counterexample takes minutes to find**
2. **Strengthening the tested property**
   - best one: “unwinding conditions” (next slide)
     - **all errors found in milliseconds, even with simple generation**
   - requires finding stronger invariants, like for proving
3. **Fancy shrinking**

1. Rather simple custom generator

```
frequency $
  [ (1,  pure Noop) ] ++
  [ (1,  pure Halt) ] ++
  [ (10,  pure Add) | nstk >= 2 ] ++
  [ (10,  Push <$> lint) ] ++
  [ (10,  pure Pop) | nstk >= 1 ] ++
  [ (20,  pure Store) | nstk >= 2
                     , absAdjustAddr vtop `isIndex` mem ] ++
  [ (20,  pure Load) | nstk >= 1
                    , absAdjustAddr vtop `isIndex` mem ] ++
  [ (10,  liftM2 Call (choose (0, (nstk-1) `min` maxArgs)) arbitrary)
                     | nstk >= 1
                     , cally ] ++
  [ (20,  liftM Return arbitrary) | Just r <- [ fmap astkReturns $
                                                 find (not . isAData) stk]
                                  , nstk >= if r then 1 else 0
                                  , cally ] ++
  [ (10,  pure Jump) | nstk >= 1
                    , jumpy ] ++
  ....
```

Guarded (add needs 2 arguments)

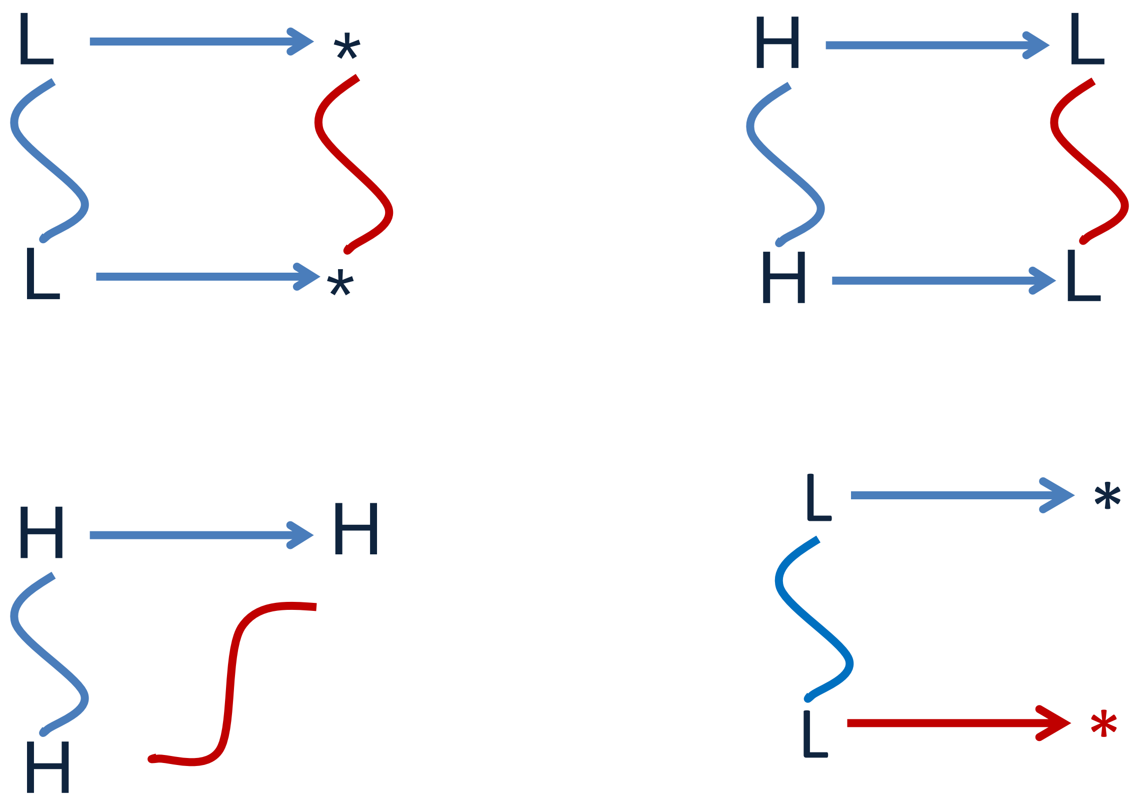
Store only to valid address

Return only after call

Biased generator

2. Stronger: Unwinding conditions

inductive invariants for noninterference are easiest to test



When should one stop? How to test the testing infrastructure?

POLARIZED MUTATION TESTING

Testing ... when should one stop?

- When testing finds no bugs
  - either there are indeed none
  - or our testing is simply not good enough
    - *"testing can only show the presence of bugs, not their absence"* – Dijkstra
- **Mutation testing**: automatically introduce realistic bugs
  - **test the testing infrastructure** (e.g. the generator)
  - in ICFP 2013 experiments we added bugs manually
    - does not scale, tedious and turns code into spaghetti
- One should one stop testing and start proving
  - when testing finds all mutants but no new bugs

Extended IFC experiment

- More realistic IFC machine
  - **extra features**: registers, public labels, dynamic allocation
  - unwinding conditions use **more complex invariants**:
    - noninterference uses stamp-based memory indistinguishability
    - H stamped regions cannot be reached through L labeled pointers
  - prior (paper) proof attempt timed out after 3 weeks of work
- Easy to enumerate all missing taints and missing checks
    - especially easy when IFC split into separate “rule table”

Rule table

	Allow	**Result**	**PC**
OpLab	TRUE	BOT	LabPC
OpMLab	TRUE	Lab1	LabPC
OpPcLab	TRUE	BOT	LabPC
OpBCall	TRUE	JOIN Lab2 LabPC	JOIN Lab1 LabPC
OpBRet	LE (JOIN Lab1 LabPC) (JOIN Lab2 Lab3)	Lab2	Lab3
OpFlowsTo	TRUE	JOIN Lab1 Lab2	LabPC
OpLJoin	TRUE	JOIN Lab1 Lab2	LabPC
OpPutBot	TRUE	BOT	LabPC
OpNop	TRUE	__	LabPC
OpPut	TRUE	BOT	LabPC
OpBinOp	TRUE	JOIN Lab1 Lab2	LabPC
OpJump	TRUE	__	JOIN LabPC Lab1
OpBNZ	TRUE	__	JOIN Lab1 LabPC
OpLoad	TRUE	Lab3	JOIN LabPC (JOIN Lab1 Lab2)
OpStore	LE (JOIN Lab1 LabPC) Lab2	Lab3	LabPC
...	...	...	...

Results encouraging

- Our **generator had tons of bugs**
  - could only kill 9 out of 51 mutants (17.6%)!
- **Finding and fixing generator bugs**
  - gathering statistics, constructing counterexamples by hand
  - fixing one generator bug usually killed many more mutants
  - sometimes found extra bugs in un-mutated artifact & property
- After a couple of days **only live 2 mutants**
  - for which we still couldn’t find counterexamples by hand
  - we applied these mutantions, started proving ... results still pending
- **Mutation testing gamifies invariant finding**
  - to the point it’s actually fun and addictive!

Mutant game (final output)

```
./Extracted
Fighting 52 mutants
Killed mutant 0 (1 frags)
Killed mutant 1 (2 frags)
Killed mutant 2 (3 frags)
Killed mutant 3 (4 frags)
Killed mutant 4 (5 frags)
Killed mutant 5 (6 frags)
Killed mutant 6 (7 frags)
Killed mutant 7 (8 frags)
Killed mutant 8 (9 frags)
Killed mutant 9 (10 frags)
Killed mutant 10 (11 frags)
Killed mutant 11 (12 frags)
Killed mutant 12 (13 frags)
Killed mutant 13 (14 frags)
Killed mutant 14 (15 frags)
Killed mutant 15 (16 frags)
Killed mutant 16 (17 frags)
Killed mutant 17 (18 frags)
Killed mutant 18 (19 frags)
Killed mutant 19 (20 frags)
Killed mutant 20 (21 frags)
Killed mutant 21 (22 frags)
Killed mutant 22 (23 frags)
Killed mutant 23 (24 frags)
Killed mutant 24 (25 frags)
Killed mutant 25 (26 frags)
Killed mutant 26 (27 frags)
Killed mutant 27 (28 frags)
Killed mutant 28 (29 frags)
Killed mutant 29 (30 frags)
Killed mutant 30 (31 frags)
Killed mutant 31 (32 frags)
Killed mutant 32 (33 frags)
Killed mutant 33 (34 frags)
Killed mutant 34 (35 frags)
Killed mutant 35 (36 frags)
Killed mutant 36 (37 frags)
Killed mutant 37 (38 frags)
Missed mutant [38] (38 frags)
Missed mutant [39] (38 frags)
Killed mutant 40 (39 frags)
Killed mutant 41 (40 frags)
Killed mutant 42 (41 frags)
Killed mutant 43 (42 frags)
Killed mutant 44 (43 frags)
Killed mutant 45 (44 frags)
Killed mutant 46 (45 frags)
Killed mutant 47 (46 frags)
Killed mutant 48 (47 frags)
Killed mutant 49 (48 frags)
Killed mutant 50 (49 frags)
Killed mutant 51 (50 frags)
```

So why did this work so well?

- Yes, human is in the loop (debugging, finding counterexamples)
  – but **we don’t waste human cycles**
- **Each unkilled mutant thought us something**
  – either exposed real bugs in the testing
  – or was apparently better than the original (more permissive)
- **This is usually not the case for mutation testing**
  – purely syntactic mutations (replace “+” by “-”)
  – human cycles wasted on silly (“equivalent”) mutants that don’t break the tested property
  – kill count just alternative to code coverage metrics, never 100%
  – what we do seems to go beyond the state of the art

Polarized mutation testing

- Generalizing this technique beyond IFC
- Started with STLC experiment
  - break progress by strengthening the step relation (e.g. dropping whole stepping rules)
  - break preservation
    - by strengthening positive occurrence of typing relation
    - or by weakening negative occurrence of typing relation
    - or by weakening (negative occurrence of) step relation
- Used Coq relational extraction plugin [Dubois et al]
- Tested against MuCheck (new Haskell mutation framework)
- No-shadowing bug in fancy generator for well-typed terms

Other experiments

- Looking at PLT Redex for already tested large formalizations
- Removed precondition for tail call optimization in CompCert
  - CSmith couldn't find the bug, despite small counterexample
  - "This is a good example to show how much more Csmith can improve"

```
#include <stdio.h>
#include <stdlib.h>

int *p;

int* bar() { // Signatures must match
  int x = 17;
  printf("%d %d\n", x, *p);
  return &x; // Need to get &x to avoid storing in register
}

int* foo() { // Signatures must match
  int q = 42;
  p = &q;
  return bar(); // Need to return for the tail call to apply
}

int main() {
  p = malloc(sizeof(int));
  foo();
  return 0;
}
```

QUICKCHECK CLONE FOR COQ

QuickCheck clone for Coq (prototype)

- ICFP 2013 work used Haskell QuickCheck
- Since then Leo ported Haskell QuickCheck to Coq
- Largest part implemented in Coq itself
- Using extraction to Haskell for
  - efficient evaluation, random seed, tracing
- At this point no big advantage over
  - writing equivalent executable spec
  - extracting it to Haskell
  - using Haskell QuickCheck

Same thing as before, just in Coq

```
Definition f x :=
match x with
| 0 => 0
| 1 => 1
| _ => 6
end%Z.

Definition prop_int_sqrt x :=
  ((x >= 0)  ==>  ((f x * f x <= x)  &&  ((f x + 1) * (f x + 1) > x)))%Z.

Definition prop_int_sqrt_small :=
  forAllShrink show (chooseZ (0%Z, 10000%Z)) shrink prop_int_sqrt.
```

```
[1 of 1] Compiling Extracted        ( Extracted.hs, Extracted.o )
Linking Extracted ...
./Extracted +RTS -K100000000 -RTS
2
Failure 1 11 0 1079681135 1 0
"*** Failed! After 1 tests and 11 shrinks" [] "Falsifiable"
```

Custom generator in Coq

```
frequency (pure Nop) [
  (* Nop *)
  (1, pure Nop);
  (* Halt *)
  (0, pure Halt);
  (* PcLab *)
  (10, liftGen PcLab genRegPtr);
  (* Lab *)
  (10, liftGen2 Lab genRegPtr genRegPtr);
  (* MLab *)
  (onNonEmpty dptr 10, liftGen2 MLab (elements Z0 dptr) genRegPtr);
  (* FlowsTo *)
  (onNonEmpty lab 10,
   liftGen3 FlowsTo (elements Z0 lab)
            (elements Z0 lab) genRegPtr);
  (* LJoin *)
  (onNonEmpty lab 10, liftGen3 LJoin (elements Z0 lab)
                                  (elements Z0 lab) genRegPtr);
  (* PutBot *)
  (10, liftGen PutBot genRegPtr);
  (* BCall *)
  (10 * onNonEmpty cptr 1 * onNonEmpty lab 1,
   liftGen3 BCall (elements Z0 cptr) (elements Z0 lab) genRegPtr);
  (* BRet *)
  (if containsRet stk then 50 else 0, pure BRet);
  (* Alloc *)
  (200 * onNonEmpty num 1 * onNonEmpty lab 1,
   liftGen3 Alloc (elements Z0 num) (elements Z0 lab) genRegPtr);
```

Some ideas about

DEEPER INTEGRATION WITH COQ/SSREFLECT

Testing actual lemmas & proof goals

- Currently
  - write executable spec in Coq
  - prove equivalence
  - test this executable variant
- Ideally, switch freely between
  - proving and testing
  - declarative and executable ...

SSReflect

- in small-scale reflection proofs
  - defining both declarative and computational specs
  - switching freely between them

... is already the normal **proving** process

- testing would add small(er) additional overhead
- SSReflect computational specifications are often not fully / efficiently executable, but
  - could use CoqEAL refinement framework [Maxime et al, ITP 2012, CPP 2013] for switching to efficiently executable code

Potential workflow

- Reify proof goal to syntactic representation of formula (Coq plugin)
- Normalize formula (DNF, classically equivalent)
- Associate computations to atoms (type classes)
  - negative atoms (premises) get **generator views**
  - positive atoms (conclusions) get **checker views**
- Associate Skolem functions to existentials (type class)
- User would still have to provide type class instances
  - could try to use existing work for partially automating this
  - full automation not our main concern, customization is

Related work (Coq)

- Sean Wilson [PhD thesis, Edinburgh, 2011]
  - qc tactic, part of larger a Coq plugin (rippling)
  - dependently-typed programming in Matthieu's Russel
  - seems rather basic, no user customization
  - only a couple of very simple examples about lists and trees
  - seems discontinued since 2011 (Coq 8.3)
- Plugins for Coq extracting inductives to ...
  - OCaml [Delahaye, Dubois, Étienne, TPHOLs 2007]
  - certified Coq [Tollitte, Delahaye, Dubois, CPP 2012]
- anything else?

THANK YOU

collaborators, CRASH/SAFE team, Amin Alipour, Johannes Borgström, Thomas Braibant, Cristian Cadar, Delphine Demange, Catherine Dubois, Matthias Felleisen, Robby Findler, Alex Groce, Rahul Gopinath, Andy Gordon, Casey Klein, Ben Karel, Scott Moore, Ulf Norell, Rishiyur S. Nikhil, Michal Palka, Manolis Papadakis, John Regehr, Howard Reubenstein, Alejandro Russo, Nick Smallbone, Deian Stefan, Greg Sullivan, Andrew Tolmach, Meng Wang, Xuejun Yang, ...