A Certified Compiler for Verifiable Computing

Cédric Fournet Chantal Keller *Vincent Laporte*

CSF — June 29th, 2016

Outsourcing computations to untrusted parties

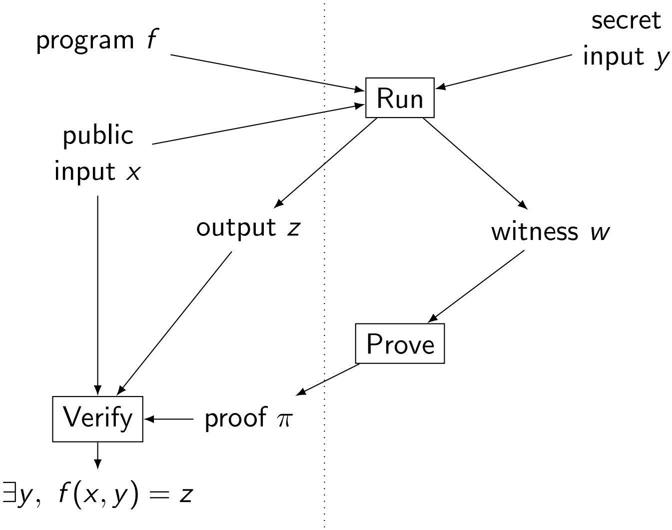
Verifier

- Wants some program to be executed
- Has limited resources

Worker

- Has the required power and secrets
- Is neither reliable, nor trusted

Pinocchio: protocols for verifiable computation



Pinocchio: practical verifiable computation

Witness: all intermediate values

- Too large
- Reveals all secrets

Proof: nine points on elliptic curves

- 288 octets (verifiable in $\sim$ 10ms)
- Look like random points

Verify: divisibility check between polynomials

- Program execution is encoded as polynomial of (very) high degree
- Proof validity expressed as a divisibility relation
- Test this relation at a single (random) point

Example program

```
int main(void) {
    int N = public-input();
    int a = private-input();
    int b = private-input();
    int r = (a != 1) * (b != 1)
          * (N == a * b);
    output(r);
    return 0;
}
```

If the output is one, then N is not prime and the worker *most probably* knows how to factor it.

Example program

```
int main(void) {
    int N = public-input();
    int a = private-input();
    int b = private-input();
    int r = (a != 1) * (b != 1)
          * (N == a * b);
    output(r);
    return 0;
}
```

$$
\begin{aligned}
(1 - x_0) \times (a - 1) &= 0 \\
x_1 \times (a - 1) &= x_0 \\
(1 - x_2) \times (b - 1) &= 0 \\
x_3 \times (b - 1) &= x_2 \\
a \times b &= x_4 \\
x_5 \times (N - x_4) &= 0 \\
x_6 \times (N - x_4) &= 1 - x_5 \\
x_0 \times x_2 &= x_7 \\
x_7 \times x_5 &= r
\end{aligned}
$$

If the output is one, then N is not prime and the worker *most probably* knows how to factor it.

QAP: System of quadratic equations in a field.

QAPs are not for programming

Pinocchio theorem

If the proof verification succeeds, then the worker most probably knows a solution to the QAP.

- Can we use a general-purpose programming language (e.g., C) and compile it into a QAP?
- Can we get similarly strong guarantees on the source program?

This work: PinocchioQ

A certified compiler for Pinocchio

- CompCert front-end from C to RTL
- A RTL *interpreter* to:
  - compile to a QAP;
  - run to produce outputs and witnesses.
- Geppetto engine to compute proofs
- A proof verifier specification

Theorem (formalized in Coq)

If the proof verification succeeds w.r.t. some public I/Os,
then there exists an execution of the *source* program with said I/Os.

The CompCert C compiler

- Optimizing C compiler to x86, PowerPC and ARM
- Various intermediate representations
- Formal semantics of all the languages involved
- Machine-checked proof of a correctness theorem

Theorem

The compiler does not introduce new behaviours:

$$\forall\ p\ p',\ \text{compile}(p) = \text{OK}(p') \implies [\![p]\!] \subseteq \text{notWrong} \implies [\![p']\!] \subseteq [\![p]\!].$$

$[\![p]\!]$: set of behaviours of program p (not empty)

Behaviours and traces

In CompCert

A behaviour is a trace (i.e., a possibly infinite list) of visible events:

- call to external function;
- volatile memory access.

Some behaviours are wrong:
reaching a (non-final) state from which the execution is stuck.

In PinocchioQ

Finite list of input or output events
(modeled through volatile memory accesses).

A modular interpreter

Three layers for independent proofs

1. **Values and arithmetic operations**
   Two implementations:
   - Symbolic values with quadratic equations (compilation)
   - Machine integers (evaluation)
2. **Memory loads/stores and pointer arithmetic**
   Resolves addresses at compile-time
3. **Control-flow and events**
   Builds an execution trace with symbolic values from the first layer.
   Resolves branches at compile-time: loops are completely unrolled

From RTL to QAP

Symbolic values

Linear combinations of wires (QAP variables).
The interpreter is stateful: remember a set of quadratic equations.

Easy case

Linear operations (add, sub): direct symbolic computation.
Multiplications: add fresh wire for the result, add equation.

Beyond multiplication, e.g., r = a != 0

Two fresh wires: q, r; two equations: $\left\{ \begin{array}{rcl} (1-r) \times a & = & 0 \\ q \times a & = & r \end{array} \right.$

Proof invariant

State only *increases*: solutions to the QAP at the end of every step are solutions to the QAP at the beginning of the step.

Range analysis

Some encodings are only valid within some range

$\begin{cases} (1-r) \times a & = & 0 \\ q \times a & = & r \end{cases}$ only works if $a \in]-2^{32}; 2^{32}[$.

Indeed 2^{32} is not null in the field, but null as 32-bit machine integer.

Range analysis

- Public inputs are annotated with a range
- For each value, we over-approximate it with an interval

Proof invariant

Assuming a solution complies with the intervals attached to public inputs, the computed ranges ensure that for every value there is exactly one integer in the range that represents this value.

Binary decomposition

Some operations have no simple quadratic encoding, e.g., right shift.
We can perform a (costly) binary decomposition and *cache* the result.

Knowing: $range(v) \subseteq [0; 2^{b+1} - 1]$

$$\begin{array}{rcl} x_0 \times (1 - x_0) & = & 0 \\ & \vdots & \\ x_b \times (1 - x_b) & = & 0 \\ v & = & \sum_{i=0}^{b} 2^i x_i \end{array}$$

Proof invariant

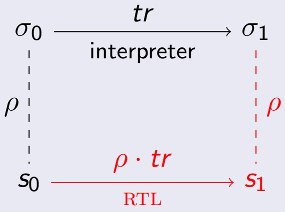
Cached binary decompositions are consistent with the decomposed value.

Simulation proof

Theorem

If the interpretation of a program P produces an symbolic trace *tr* and QAP q, if ρ is a solution to the QAP, then the concrete trace $\rho \cdot tr$ is a bahaviour of P.

Simulation lemma



Conclusions in red.

σ_0, σ_1, interpreter states (QAP)

ρ solution to σ_1

s_0, s_1, RTL states

σ_0 and s_0 are related (given ρ)

Then induction on the length of the interpretation.

Proof verification

- A divisibility check between polynomials
- New Coq library for Lagrange polynomials
- Formal link between the divisibility relation and QAP solutions

Theorem

If the divisibility relation between the polynomials constructed from the QAP and its putative solution holds, then it is an actual solution to the QAP.

Experimental evaluation

- Extract and run our compiler on various C programs
- Connect to the Geppetto engine for the cryptography part

Case	RTL instructions		Size (#wire)		Degree	Compile (s)		Evaluate (s)		KeyGen	Prove	Verify
	static	dynamic	I/O	private		partial	total	partial	total	(s)	(s)	(ms)
First	21	34	4	2	3	0.00	0.00	0.00	0.00	0.03	0.01	10
Factorization	40	167	2	19	20	0.00	0.01	0.00	0.01	0.04	0.01	12
Bachet	63	527	2	64	65	0.00	0.02	0.00	0.01	0.08	0.02	12
Matrix (10)	97	37 892	201	1800	1900	0.22	0.37	0.15	0.18	0.90	0.34	13
Matrix (100)	97	17 251 542	20 001	1 080 000	1 090 000	143.72	178.62	66.34	68.37	229.37	247.70	228
SHA1 (4)	180	29 313	7	36 138	37 082	4.59	5.65	0.14	0.28	25.82	49.75	11
SHA1 (96)	180	58 831	30	77 004	78 925	10.02	12.26	0.29	0.45	26.33	32.98	11
SHA1 (159)	180	88 251	46	116 325	119 209	15.33	18.55	0.43	0.62	32.56	49.83	11
SAT (20)	39 462	40 262	1	4220	4220	0.22	9.27	0.72	9.01	1.46	1.25	14
SAT (50)	583 902	588 902	1	63 800	63 800	5.67	1843.5	12.42	1842.9	14.07	18.81	12

Conclusion

Summary

- Coq implementation and proof of a compiler from C to QAP, based on CompCert
- Linked to the Geppetto engine, complete implementation of the Pinocchio protocol (soon available at `https://vc.codeplex.com/`)
- Formal link between the proof verification and the execution of the source program

Future work

- Verify the implementation of the proof verifier
- Bootstrapping