The security protocol verifier ProVerif and its recent improvements: lemmas, induction, fast subsumption, and much more

Bruno Blanchet

Inria, Paris, France
Bruno.Blanchet@inria.fr

joint work with Vincent Cheval and Véronique Cortier

May 2022

Cryptographic protocols

(1)

Cryptographic protocols

- small programs designed to secure communication (various security goals)
- use cryptographic primitives (e.g. encryption, hash function, ...)

(1) by Fabio Lanari — Internet1.jpg by Rock1997 modified., CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=20995390

Models of protocols

Active attacker:

- The attacker can intercept all messages sent on the network
- He can compute messages
- He can send messages on the network

The symbolic model

The symbolic model or "Dolev-Yao model" is due to Needham and Schroeder (1978) and Dolev and Yao (1983).

- Cryptographic primitives are blackboxes. senc
- Messages are terms on these primitives. senc($Hello$, k)
- The attacker is restricted to compute only using these primitives.
  ⇒ perfect cryptography assumption
  - So the definitions of primitives specify what the attacker can do.
    One can add equations between primitives.
    Hypothesis: the only equalities are those given by these equations.

This model makes automatic proofs relatively easy.

Features of ProVerif

- Fully automatic.
- Works for unbounded number of sessions and message space.
  - $\Rightarrow$ undecidable problem
- Handles a wide range of cryptographic primitives, defined by rewrite rules or equations.
- Handles various security properties: secrecy, authentication, some equivalences.
- Does not always terminate and is not complete. In practice:
  - Efficient: small examples verified in less than 0.1 s;
    complex ones in a few minutes.
  - Very precise: no false attack in our tests on examples of the literature for secrecy and authentication.

ProVerif, `https://proverif.inria.fr/`

Protocol: Pi calculus + cryptography Primitives: rewrite rules, equations	Properties to prove: Secrecy, authentication, process equivalences

↓

Automatic translator

↓

Horn clauses	Derivability queries

↓

Resolution with selection

↙ Non-derivable: the property is true

↘ Derivation

- ↙ Attack: the property is false
- ↘ False attack: I don't know

Syntax of the process calculus

Pi calculus + cryptographic primitives

$M, N ::=$	terms
$x, y, z, \ldots$	variable
$a, b, c, s, \ldots$	name
$f(M_1, \ldots, M_n)$	constructor application
$P, Q ::=$	processes
$\textbf{out}(M, N); P$	output
$\textbf{in}(M, x : T); P$	input
0	nil process
$P \mid Q$	parallel composition
$!P$	replication
$\textbf{new}\ a : T; P$	restriction
$\textbf{let}\ x = g(M_1, \ldots, M_n)\ \textbf{in}\ P\ \textbf{else}\ Q$	destructor application
$\textbf{if}\ M = N\ \textbf{then}\ P\ \textbf{else}\ Q$	conditional

Constructors and destructors

Two kinds of operations:

- Constructors f are used to build terms: $f(M_1, \dots, M_n)$

Example: Shared-key encryption senc(M, N)

fun senc(bitstring, key) : bitstring.

- Destructors g manipulate terms: **let** $x = g(M_1, \dots, M_n)$ **in** P **else** Q
  Destructors are defined by rewrite rules $g(M_1, \dots, M_n) \rightarrow M$.

Example: Decryption sdec(senc$(m, k), k) \rightarrow m$

fun sdec(bitstring, key) : bitstring
reduc forall m : bitstring, k : key; sdec(senc$(m, k), k) = m$.

We represent in the same way public-key encryption, signatures, hash functions, ...

Example: The Denning-Sacco protocol (simplified)

$$
\begin{array}{llll}
\text{Message 1.} & A \rightarrow B: & \{\{k\}_{sk_A}\}_{pk_B} & k \text{ fresh} \\
\text{Message 2.} & B \rightarrow A: & \{s\}_k &
\end{array}
$$

$$
\begin{array}{lll}
& & \textbf{new } sk_A : \mathrm{sskey}; \textbf{new } sk_B : \mathrm{eskey}; \textbf{let } pk_A = \mathrm{spk}(sk_A) \textbf{ in} \\
& & \textbf{let } pk_B = \mathrm{pk}(sk_B) \textbf{ in out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & & \quad !\, \textbf{in}(c, x\_pk_B : \mathrm{epkey}); \textbf{new } k : \mathrm{key}; \\
& & \quad \textbf{out}(c, \mathrm{penc}(\mathrm{sign}(k, sk_A), x\_pk_B)); \\
& & \quad \textbf{in}(c, x : \mathrm{bitstring}); \textbf{let } s = \mathrm{sdec}(x, k) \textbf{ in } 0 \\
(B) & \mid & \quad !\, \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let } y' = \mathrm{pdec}(y, sk_B) \textbf{ in} \\
& & \quad \textbf{let } k = \mathrm{checksign}(y', pk_A) \textbf{ in out}(c, \mathrm{senc}(\mathrm{s}, k))
\end{array}
$$

The Horn clause representation

The first encoding of protocols in Horn clauses was given by Weidenbach (1999).

The main predicate used by the Horn clause representation of protocols is att:

$\mathrm{att}(M)$ *means* *"the attacker may have M".*

We can model actions of the attacker and of the protocol participants thanks to this predicate.

Processes are automatically translated into Horn clauses (joint work with Martín Abadi).

Coding of primitives

- Constructors $f(M_1, \ldots, M_n)$
  $\mathrm{att}(x_1) \wedge \ldots \wedge \mathrm{att}(x_n) \rightarrow \mathrm{att}(f(x_1, \ldots, x_n))$

Example: Shared-key encryption $\mathrm{senc}(m, k)$

$\mathrm{att}(m) \wedge \mathrm{att}(k) \rightarrow \mathrm{att}(\mathrm{senc}(m, k))$

- Destructors $g(M_1, \ldots, M_n) \rightarrow M$
  $\mathrm{att}(M_1) \wedge \ldots \wedge \mathrm{att}(M_n) \rightarrow \mathrm{att}(M)$

Example: Shared-key decryption $\mathrm{sdec}(\mathrm{senc}(m, k), k) \rightarrow m$

$\mathrm{att}(\mathrm{senc}(m, k)) \wedge \mathrm{att}(k) \rightarrow \mathrm{att}(m)$

Coding of a protocol

If a principal A has received the messages $M_1, \dots, M_n$ and sends the message M,

$$\mathrm{att}(M_1) \wedge \dots \wedge \mathrm{att}(M_n) \rightarrow \mathrm{att}(M).$$

Example

Upon receipt of a message of the form $\mathrm{penc}(\mathrm{sign}(y, sk_A), pk_B)$,
B replies with $\mathrm{senc}(\mathrm{s}, y)$:

$$\mathrm{att}(\mathrm{penc}(\mathrm{sign}(y, sk_A), pk_B)) \rightarrow \mathrm{att}(\mathrm{senc}(\mathrm{s}, y))$$

The attacker sends $\mathrm{penc}(\mathrm{sign}(y, sk_A), pk_B)$ to B, and intercepts his reply $\mathrm{senc}(\mathrm{s}, y)$.

Proof of secrecy

Theorem (Secrecy)

If $\mathrm{att}(M)$ *cannot be derived from the clauses, then* M *is secret.*

The term M cannot be built by an attacker.

The resolution algorithm will determine whether a given fact can be derived from the clauses.

Example

query attacker(s).

Resolution with free selection

$$\frac{R = H \rightarrow F \qquad R' = F'_1 \wedge H' \rightarrow F'}{H\sigma \wedge H'\sigma \rightarrow F'\sigma}$$

where σ is the most general unifier of F and F'_1,
F and F'_1 are selected.

The selection function selects:

- a hypothesis not of the form $\text{att}(x)$ if possible,
- the conclusion otherwise.

Key idea: avoid resolving on facts $\text{att}(x)$.

Resolve until a fixpoint is reached.
Keep clauses whose conclusion is selected.

Theorem

The obtained clauses derive the same facts as the initial clauses.

Other security properties (1)

Correspondence assertions (authentication):

If an event has been executed, then some other events must have been executed.

$$
\begin{array}{lll}
 & & \textbf{new } sk_A : \mathrm{sskey}; \textbf{new } sk_B : \mathrm{eskey}; \textbf{let } pk_A = \mathrm{spk}(sk_A) \textbf{ in} \\
 & & \textbf{let } pk_B = \mathrm{pk}(sk_B) \textbf{ in out}(c, pk_A); \textbf{out}(c, pk_B); \\
(A) & & \quad ! \, \textbf{in}(c, x\_pk_B : \mathrm{epkey}); \textbf{new } k : \mathrm{key}; \textbf{event } eA(pk_A, x\_pk_B, k); \\
 & & \quad \textbf{out}(c, \mathrm{penc}(\mathrm{sign}(k, sk_A), x\_pk_B)); \\
 & & \quad \textbf{in}(c, x : \mathrm{bitstring}); \textbf{let } s = \mathrm{sdec}(x, k) \textbf{ in } 0 \\
(B) & | & \quad ! \, \textbf{in}(c, y : \mathrm{bitstring}); \textbf{let } y' = \mathrm{pdec}(y, sk_B) \textbf{ in} \\
 & & \quad \textbf{let } k = \mathrm{checksign}(y', pk_A) \textbf{ in event } eB(pk_A, pk_B, k); \\
 & & \quad \textbf{out}(c, \mathrm{senc}(s, k))
\end{array}
$$

$$\textbf{query } x : \mathrm{spkey}, y : \mathrm{epkey}, z : \mathrm{key}; \textbf{event}(eB(x, y, z)) \Longrightarrow \textbf{event}(eA(x, y, z))$$

Other security properties (2)

Process equivalences:

- Strong secrecy: the attacker cannot distinguish when the value of the secret changes.
- diff-equivalence: Equivalence between processes that differ only by terms they contain (joint work with Martín Abadi and Cédric Fournet)

  In particular, proof of protocols relying on weak secrets.

Extensions

1. Natural numbers
2. Temporal correspondence queries
3. Precise actions
4. Axioms, Restrictions, Lemmas
5. Proofs by induction

Natural numbers

- Type: `nat`
- Allowed operations:
  - addition, subtraction between variable and natural number
  - less, less or equal, greater, greater or equal
  - predicate testing if a term is a natural number: is_nat

```
free k:key [private]. free cell:channel [private].

(* outputs natural numbers from min to max encrypted with k *)
let Q(max:nat) =
  in(cell,i:nat); out(c,senc(i,k));
  if i < max then out(cell,i+1).

process in(c, (min:nat, max:nat));
        (out(cell,min) | !Q(max))
```

Implemented by constraints is_nat(M), $\neg$is_nat(M), and $M \geq N + n$ in clauses, where n is a constant natural number, simplified using the Bellman-Ford algorithm.

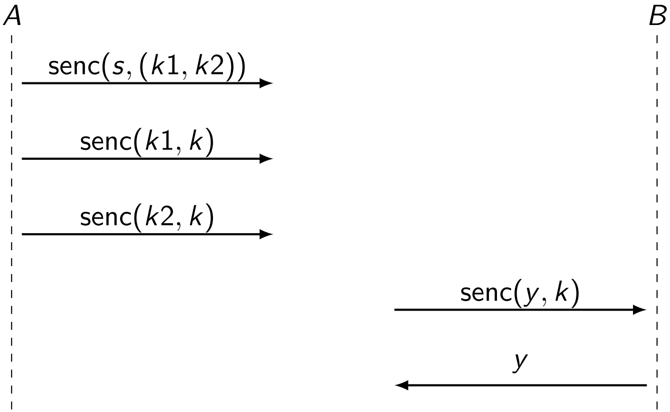
Temporal correspondence queries

- Type `time` for temporal variables.
- Facts can be associated with a temporal variable: $F@i$.
- **event**(ev)@n holds when event ev is executed at the n-th step of the trace.
- Can compare temporal variables:

```
query i,j:time, x:bitstring;
  event(A(x))@i && event(B(x))@j ==> i < j.
```

- Encoded as special natural number constraints $i < j$ and $i \leq j$.

Precise actions: toy example



B acts as an oracle for decryption with the key k but only one time!

Precise actions: process and clauses

Process

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring);
  out(c,sdec(x,k)).

process A | B
```

Clauses

- for the process
  - A:

$\mathrm{att}(\mathrm{senc}(s,(k_1,k_2)))$
$\mathrm{att}(\mathrm{senc}(k_1,k))$
$\mathrm{att}(\mathrm{senc}(k_2,k))$

  - B:

$\mathrm{att}(\mathrm{senc}(y,k)) \rightarrow \mathrm{att}(y)$

- for the attacker

$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(\mathrm{senc}(x,y))$
$\mathrm{att}(\mathrm{senc}(x,y)) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(x)$
$\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}((x,y))$

Secrecy of s is proved when $\mathrm{att}(s)$ is not derivable from the clauses.

Precise actions: why does it fail?

Process

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring);
  out(c,sdec(x,k)).

process A | B
```

Clauses

– for the process

Horn clauses can be applied an arbitrary number of times for arbitrary instantiations

att(senc(k_2, k))

– B:

att(senc(y, k)) $\rightarrow$ att(y)

– for the attacker

att(x) $\wedge$ att(y) $\rightarrow$ att(senc(x, y))
att(senc(x, y)) $\wedge$ att(y) $\rightarrow$ att(x)
att(x) $\wedge$ att(y) $\rightarrow$ att((x, y))

Secrecy of s is proved when att(s) is not derivable from the clauses.

Precise actions: why does it fail?

$$
\begin{tikzcd}
 & \boxed{\mathrm{att}(\mathrm{senc}(k_1, k))} \arrow[d, "\mathrm{att}(\mathrm{senc}(k_1, k))"'] & & \boxed{\mathrm{att}(\mathrm{senc}(k_2, k))} \arrow[d, "\mathrm{att}(\mathrm{senc}(k_2, k))"'] \\
 & \boxed{\mathrm{att}(\mathrm{senc}(y, k)) \rightarrow \mathrm{att}(y)} \arrow[dr, "\mathrm{att}(k_1)"'] & & \boxed{\mathrm{att}(\mathrm{senc}(y, k)) \rightarrow \mathrm{att}(y)} \arrow[dl, "\mathrm{att}(k_2)"] \\
\boxed{\mathrm{att}(\mathrm{senc}(s, (k_1, k_2)))} \arrow[dr, "\mathrm{att}(\mathrm{senc}(s, (k_1, k_2)))"'] & & \boxed{\mathrm{att}(x) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}((x, y))} \arrow[dl, "\mathrm{att}((k_1, k_2))"] & \\
 & \boxed{\mathrm{att}(\mathrm{senc}(x, y)) \wedge \mathrm{att}(y) \rightarrow \mathrm{att}(x)} \arrow[d, "\mathrm{att}(s)"'] & & \\
 & {} & &
\end{tikzcd}
$$

Precise actions: what to do?

- Add a [precise] option to the problematic input.

```
free s,k1,k2,k:bitstring [private].

let A =
  out(c,senc(s,(k1,k2)));
  out(c,senc(k1,k));
  out(c,senc(k2,k)).

let B =
  in(c,x: bitstring) [precise];
  out(c,sdec(x,k)).

process A | B
```

- Global setting: `set preciseActions = true.`
- Adding [precise] options may increase the verification time or lead to non-termination.

Restrictions, axioms, lemmas

```
restriction R1 .
...
restriction Rn .

axiom A1 .
...
axiom Am .

lemma L1 .
...
lemma Lk .

query attacker(s).
```

Restrictions "restrict" the traces considered in axioms, lemmas, and queries.
query attacker(s) holds if no trace satisfying $R_1, \dots, R_n$ reveals s.

1. ProVerif assumes that the axioms $A_1, \dots, A_m$ hold.
2. ProVerif tries to prove the lemmas $L_1, \dots, L_k$ in order, using all axioms and previously proved lemmas.
3. ProVerif tries to prove the query **query attacker**(s) using all axioms and all lemmas.

Implementing precise actions

Option [precise] is encoded as an axiom internally.

```
let B =
  in(c,x: bitstring) [precise];
  out(c,sdec(x,k)).
```

encoded as

```
event Precise(occurrence,bitstring).

axiom occ:occurrence, x1,x2:bitstring;
  event(Precise(occ,x1)) && event(Precise(occ,x2)) ==> x1 = x2.

let B = in(c,x:bitstring);
        new occ[]:occurrence;
        event Precise(occ,x);
        out(c,sdec(x,k)).
```

Using restrictions, axioms, and lemmas (simplified)

Consider a lemma (or restriction or axiom) $\bigwedge_i F_i \implies \phi$.

$$\frac{H \rightarrow C \qquad \text{for all } i, F_i\sigma \in H \text{ or } F_i\sigma = C}{H \wedge \phi\sigma \rightarrow C}$$

If for all i, $F_i\sigma \in H$ or $F_i\sigma = C$, then the hypothesis of the lemma holds, so the conclusion of the lemma holds. We add it to the hypothesis of the clause, generating clause $H \wedge \phi\sigma \rightarrow C$.

Example

Axiom $\mathbf{event}(\mathit{Precise}(\mathit{occ}, x_1)) \wedge \mathbf{event}(\mathit{Precise}(\mathit{occ}, x_2)) \implies x_1 = x_2$.

$\mathbf{event}(\mathit{Precise}(\mathit{occ}, \mathrm{senc}(k_1, k))) \wedge \mathbf{event}(\mathit{Precise}(\mathit{occ}, \mathrm{senc}(k_2, k))) \rightarrow \mathrm{att}(s)$

transformed into

$\mathbf{event}(\mathit{Precise}(\mathit{occ}, \mathrm{senc}(k_1, k))) \wedge \mathbf{event}(\mathit{Precise}(\mathit{occ}, \mathrm{senc}(k_2, k))) \wedge$
$\quad \mathrm{senc}(k_1, k) = \mathrm{senc}(k_2, k) \rightarrow \mathrm{att}(s)$

Removed.

Proofs by induction

- In order to prove a query, use that query itself as lemma on a strict prefix of the trace, by induction on the length of the trace.
- In a clause $H \rightarrow C$, H happens strictly before C.
- Consider the inductive lemma $\bigwedge_i F_i \implies \phi$.
  ϕ holds before or at the same time as the latest F_i.

$$\frac{H \rightarrow C \qquad \text{for all } i, F_i\sigma \in H}{H \wedge \phi\sigma \rightarrow C}$$

  If for all i, $F_i\sigma \in H$, then the hypothesis of the lemma holds strictly before C, so the conclusion of the lemma holds strictly before C. We add it to the hypothesis of the clause, generating clause $H \wedge \phi\sigma \rightarrow C$.
- Also works for a group of queries: proofs by mutual induction.

Proofs by induction: example

```
free cell:channel [private].

query x:nat;
mess(cell,x)==>is_nat(x).

let Q =
  in(cell,i:nat);
  out(c,senc(i,k));
  out(cell,i+1).

process out(cell,0) | !Q
```

Clauses:

$\mathrm{mess}(\mathit{cell}, 0)$

$\mathrm{mess}(\mathit{cell}, i) \rightarrow \mathrm{mess}(\mathit{cell}, i + 1)$

ProVerif stops resolving on $\mathrm{mess}(\mathit{cell}, i)$ because it would lead to an infinite loop.

The attacker is untyped: a priori, i may not be a natural number.

The proof fails.

Proofs by induction: example solved

```
free cell:channel [private].

set nounifIgnoreAFewTimes = auto.

query x:nat;
mess(cell,x)==>is_nat(x) [induction].

let Q =
  in(cell,i:nat);
  out(c,senc(i,k));
  out(cell,i+1).

process out(cell,0) | !Q
```

Clauses:

$\text{mess}(cell, 0)$

$\text{mess}(cell, i) \rightarrow \text{mess}(cell, i+1)$

Lemma $\text{mess}(cell, x) \Longrightarrow \text{is\_nat}(x)$
transforms
$\text{mess}(cell, i) \rightarrow \text{mess}(cell, i+1)$
into
$\text{mess}(cell, i) \wedge \text{is\_nat}(i) \rightarrow \text{mess}(cell, i+1)$

nounifIgnoreAFewTimes allows resolution on $\text{mess}(cell, i)$ once during verification.

The proof now succeeds.

Expressivity results

- P Precise actions
- I `set nounifIgnoreAFewTimes = auto.`
- R `set removeEventsForLemma = true.`
  Remove events used only for lemmas, when they become useless.
- ℕ Natural numbers
- A Axioms, Lemmas

Expressivity results

Published protocols

Protocol	Q	O	#	N	P	I	R	ℕ	A
PCV Otway-Rees	eq	✗	1	✓	◆				
PCV Needham-Schreder	inj	✗	6	✓	◆	◆			
			3	ϟ					
PCV Denning-Sacco	inj	✗	1	ϟ					
JFK	cor	✗	2	ϟ		◆			
	inj		2	✓					
Arinc823	cor	✗	6	ϟ				◆	
Helios-norevote	eq	✗	4	✓	◆				
Signal	cor	✗	2	ϟ					
TLS12-TLS13-draft18	cor	✗	1	ϟ					

Unpublished protocols

Protocol	Q	O	#	N	P	I	R	ℕ	A
QBC_4qbits	cor	✗	1	✓	◆				
			1	ϟ					
Voting-draft	eq	✗	1	✓	◆				
LAK-simplified	cor	⏲	1	✓		◆			
PACE_v3-sequence	cor	✗	1	✓	◆			◆	◆
			3	ϟ					
DP-3T-simpl-draft	cor	⏲	1	✓	◆	◆	◆	◆	◆
			2	ϟ					
student1	cor	⏲	2	✓	◆		◆		
	inj		1	ϟ					
student2	inj	✗	1	✓					
student3	cor	⏲	1	ϟ	◆		◆		
student4	cor	✗	2	ϟ	◆		◆		
student5	cor	⏲	1	ϟ					

Improved efficiency

- A Subsumption
- B Translation of processes into clauses
- C Resolution
- D Global redundancy
- E Pre-treatment of processes

A Subsumption

$H \rightarrow C$ subsumes $H' \rightarrow C'$ when $C\sigma = C'$ and $H\sigma \subseteq H'$.

Every time a clause is generated by resolution,

- check if it is not subsumed by an existing clause
- remove all existing clauses that are subsumed by this new clause

More than 80% of total execution time!

Idea [Schulz13]: Feature vertex indexing

A feature is a function f on clauses such that
$H \rightarrow C$ subsumes $H' \rightarrow C'$ implies $f(H \rightarrow C) \leq f(H' \rightarrow C')$

Clauses are organized in a trie indexed by feature values.

C Resolution

Resolution: One clause against many!

The selection function guarantees that always the same fact of a clause will be used.

Clauses are organized in a trie indexed by the symbol functions of their selected fact (depth first exploration)
[Substitution tree indexing techniques]

Advantage:

- Fewer unifications
- We know quickly with which clauses we can perform resolution

D Global redundancy

A clause is redundant when it is obtained by resolving existing clauses whose conclusion is selected.

1. Avoid testing redundancy when it is useless.
2. Simplified the test (e.g. subsumption is useless).

B Translation of processes into clauses

We evaluate an argument of a function only when it is still needed in order to determine the result.

Example

$M \wedge N$: if M evaluates to false, we do not evaluate N.

E Pre-treatment of processes

ProVerif sometimes groups sequences of lets

$$\textbf{let } x_1 = M_1 \textbf{ in } \ldots \textbf{let } x_n = M_n \textbf{ in } P$$

to evaluate all of $M_1, \ldots, M_n$ and then evaluate P when none of them fails.

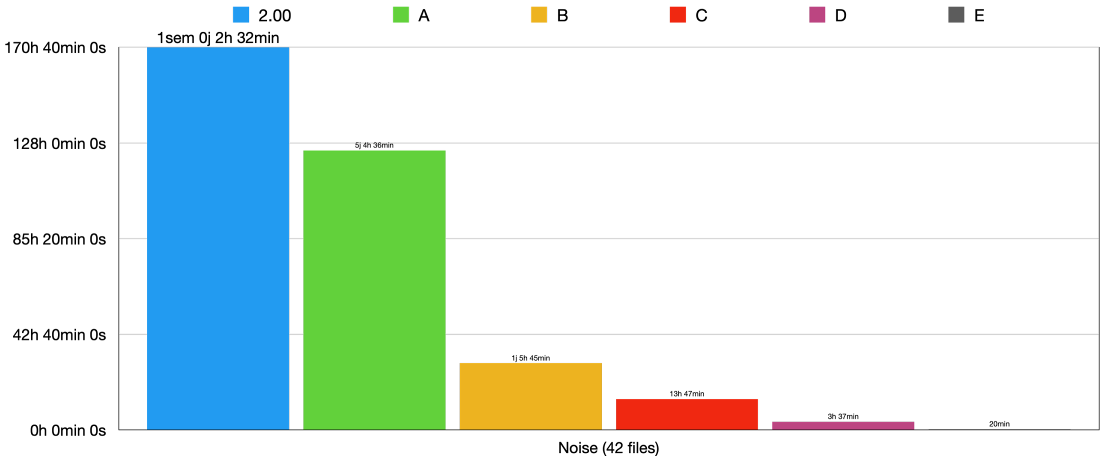
Improves precision for equivalence proofs: avoids distinguishing which M_i fails.

We ensure that M_i is not evaluated when a previous M_j fails, while keeping the improved precision.

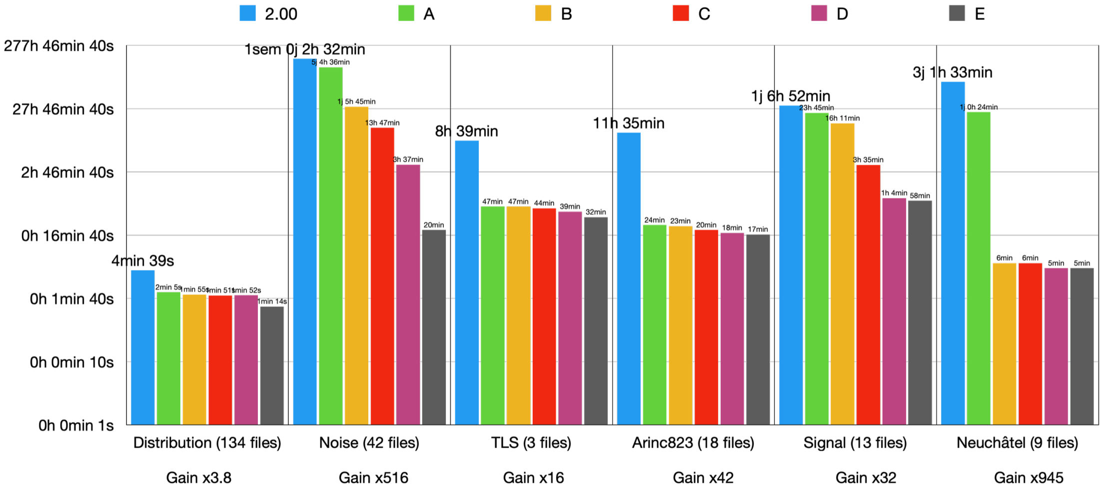
Improved efficiency

- ProVerif 2.00
- A Subsumption
- B Translation of processes into clauses
- C Resolution
- D Global redundancy
- E Pre-treatment of processes

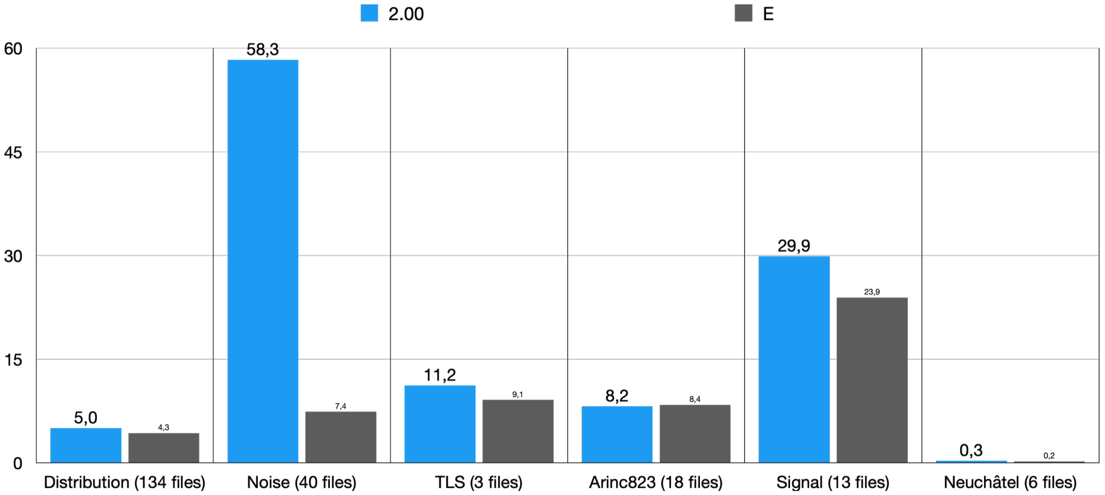
Time gain (linear scale)



Time gain (log scale)



Memory gain (linear scale, Gb)



What's next?

1. Integration of GSVerif
   - Precise actions of GSVerif much stronger than the one of ProVerif
   - New transformations?
2. Modulo AC / XOR / groups
   - The algorithm should remain mostly the same
   - Main issues : Efficiency and non-termination
3. Going beyond diff-equivalence
   - Trace equivalence
4. Whatever users need!

Paper to appear at IEEE Security and Privacy 2022

`https://bblanche.gitlabpages.inria.fr/publications/BlanchetEtAlSP22.html`