Multi-System Reasoning in the Squirrel Proof Assistant

Report of project supervised by David Baelde and Joseph Lallemand

Benjamin Voisin
ENS Rennes
Rennes, France
benjamin.voisin@ens-rennes.fr

***Abstract*— Given the central importance of designing secure protocols, providing solid mathematical foundations and computer-assisted methods to attest for their correctness is becoming crucial.**

Here, we elaborate on the Squirrel Proof Assistant by formalizing multi-system reasonning, and then by implementing multi-system proofs with more than 2 systems in the Squirrel Proof Assistant.

***Index terms*—Security Protocols, Formal Methods, Computational Security, Interactive Theorem Prover, Multi Systems**

I. Introduction

Providing solid mathematical proofs is essential in order to avoid having payment protocols allowing an attacker to steal any amount of money from a locked phone [1]. To prevent this, some automated and interactive tools such as Tamarin [2] and ProVerif [3] have been proposed to prove security of protocols.

Multiple approach exists for this goal using automated or interactive tools.

Most of them lies in the *symbolic model*, often called the "Dolev-Yao model", due to Dolev and Yao [4], where the attacker is modelised by the rules he can apply (for example, if he knows a message and its decryption key, he can decrypt it). This method can find many attacks, and allows for easy automatic proofs. However, this model is not very realistic, and can miss some attacks [5].

In the meantime, the *computational model* [6], [7] was devlopped. In this model, the messages are just bitstrings, and cryptographic primitives are functions over the bitstrings. The attacker is moddeled by a probabilisitc Turing machine, and security properties are said to hold when the probability that it is false is negligeable w.r.t. a security parameter (e.g. the size of a key) [5].

Instead of describing what the attacker can do, the computational model specify what the attacker cannot do (more formally, what is very unlikly to be done by the attacker w.r.t. the security parameter).

This model is more realistic and is the one generally used by cryptographers. However, automatic proofs are harder to do, and even assisted proofs were not done until recently.

The Squirrel Prover is an interactive theorem prover in the computationnal model [8]. It expands on the approach givien by Bana and Comon [9], [10] to allow mecanization of proofs in an interactive theorem prover.

The Squirrel Prover works by formally describing protocols, and then interactively proving properties over said protocols. There are two kinds of such properties :

- Trace properties e.g. only allowed users are accepted by the protocol.
- Privacy properties, meaning that the protocol does not leak unwanted information.

In order to prove the latter, we often compare the real protocol with an idealized one, where private information is replaces by random value, and then prove that the real and idealized protocols are equivalent for the attacker. This means that the attackers cannot distinguish the private information from a random bitstring, so it doesn't know the secret information.

So, the Squirrel Prover allow using the `diff` operator to easily describe processes that are very similar and differs in a few places. It is then possible to write once the protocol, and put a secret key under a `diff` alongside a randow value, to prove that this information is not leaked.

However, the tool currently only accepts bi-systems : processes with only 2 variants, hard-coded to be the `left` and `right` variants. This can be annoying as some proofs become very repetitive, lemmas and theorem need to be duplicated for each variant.

Another shortcomming of multi-system operations in the Squirrel Prover is their lack of proper theorical formalisation. They are not well-defined, and there is no formal background for the operations done by the tool. This has been the source of incorrect behaviour in the tool.

This report contains two main contributions : the formalisation of a new logic that allow reasonning on multi-systems, with some results over what can and cannot be done in multi-system reasonning, and the generalization of the `diff` operation in the Squirrel Prover, to allow for any number of processes.

This report is split in three sections :

- First, theory
- Then, implementation
- Finally, the conclusion followed by some perpsectives

II. The new multi-system logic

A. Terms

The base block of this logic is the term. They are very similar to first-order logic, with the addition of the `diff` operator.

We use τ to represent types. In practice, the types are either "message" or "boolean", and l to represent labels (indentifiers of projections).

Definition 1 : Inductive definition of the terms

$$t ::= x \mid f(t_1, ..., t_n) \mid \forall x : \tau \cdot t \mid \exists x : \tau \cdot t \mid \text{diff}(l_1 : t_1, ..., l_n : t_n) \tag{1}$$

The `diff` operator assign some tems to some projections. For example, the term $\text{diff}(l_1 : x, l_2 : y)$ is a term such that, its value on label l_1 is x, and its value on label l_2 is y.

We want to be able to answer the question "What is the value of the term t under the label l", but this is not always easy. For example with imbricated diffs (i.e. a term like $\text{diff}(l_1 : \text{diff}(l_3 : x, l_4 : y), l_2 : x)$

So, we will use a typing system to ensure that terms are "well-formed" :

Definition 2 : Typing system

$$\frac{\Gamma(x) = \tau^{\vec{l}}}{\Gamma \vdash x : \tau^{\vec{l}}}$$

$$\frac{\Gamma \vdash t_i : \tau_i^{\vec{l}} \quad \Gamma(f) = \tau_1 \to ... \to \tau_n \to \tau'}{\Gamma \vdash f(t_1, ..., t_n) : \tau'^{\vec{l}}}$$

$$\frac{... \qquad \Gamma \vdash t_i : \tau^{l_i} \qquad ...}{\Gamma \vdash \text{diff}(l_1 : t_1, ..., l_n : t_n) : \tau^{l_1, ..., l_n}} \tag{2}$$

$$\frac{\Gamma \vdash t : \tau^{l_1, ..., l_n}}{\Gamma \vdash t : \tau^{l'_1, ..., l'_k}} \text{ with } \{l'_1, ..., l'_k\} \subseteq \{l_1, ..., l_n\}$$

$$\frac{\Gamma(x) = \tau^{\vec{L}} \vdash t : \text{bool}^{\vec{L}}}{\Gamma \vdash (\forall x : \tau \cdot t) : \text{bool}^{\vec{L}}} \qquad \frac{\Gamma(x) = \tau^{\vec{L}} \vdash t : \text{bool}^{\vec{L}}}{\Gamma \vdash (\exists x : \tau \cdot t) : \text{bool}^{\vec{L}}}$$

We see that the term $\text{diff}(l_1 : \text{diff}(l_3 : x, l_4 : y), l_2 : x)$ cannot be typed by our typing system, as the term $\text{diff}(l_3 : x, l_4 : y)$ will be of type τ^{l_3, l_4}.

However, (and opposit to what is implemented in the Squirrel Prover tool), we can still write imbricated `diffs` like $\text{diff}(l_1 : \text{diff}(l_1 : x, l_2 : y), l_2 : y)$, but this is not an issue as the second label of the inner `diff` will never be used, the term can be simplified as $\text{diff}(l_1 : x, l_2 : y)$. They are theorically writeable, but definitely useless.

With this, it is now possible to build a "projection" operator, that gives the value of a term t on a label l_i if $\Gamma \vdash t : \tau^{l_i}$:

Definition 3 : Projection operator

$$\begin{aligned} t\#l_i = \\ &\mid x\#l_i && := x \\ &\mid f(t_1, ..., t_n)\#l_i && := f(t_1\#l_i, ..., t_n\#l_i) \\ &\mid (\forall x \cdot t)\#l_i && := \forall x \cdot (t\#l_i) \\ &\mid (\exists x \cdot t)\#l_i && := \exists x \cdot (t\#l_i) \\ &\mid \text{diff}(l_1 : t_1, ..., l_n : t_n) && := t_i\#l_i \end{aligned} \tag{3}$$

Theorem 1 : Substition theorem

Given a term t_1, if t_2 has the good type, then we can substitute x in t_1 with t_2

We can prove some usefull properties :

Theorem 2 : Type augmentation

If $\Gamma \vdash t : \tau^{l_1}, ..., \Gamma \vdash t : \tau^{l_n}$ then $\Gamma \vdash t : \tau^{l_1, ..., l_n}$

B. Formulas

We can then define formulas over the terms. They add the new operators `[]` and `equiv`.

Intuitively, the `[]` operator allow to consider a term of type `bool` on some labels $\vec{l}$ in the formulas scope.

`equiv` is the equivalence operator. Intuitively, it is true if an attacker cannot distinguish $t\#l_1$ from $t\#l_2$. The term in the equivalence operator must be of type τ^{l_1, l_2} for it to be well-defined.

Even if the `diffs` are general enough to work with any number of labels, we only consider binary equiv here, as it reflects the tool. It might be usefull to have more general `equiv`, but it is not the scope of this report.

We define formally the formulas of our multi-system logic :

Definition 4 : Inductive definition of the formulas

$$F ::= \top \mid \bot \mid F_1 \wedge F_2 \mid F_1 \Rightarrow F_2 \mid [t]^{l_1, ..., l_n} \mid \text{equiv}^{(l_1, l_2)}(t) \tag{4}$$

We deliberetly chose to not add quantifiers in the formulas scope, as they already are present within the terms, and thus are not needed to fully express what is needed in the Squirrel Prover.

We can then prove some usefull theorems for our formulas, that will be usefull to implement multi-system reasonning in the Squirrel Prover.

Theorem 3 : Inference rules over formulas

$$\frac{[\varphi]^{l_1,\dots,l_n}}{[\varphi]^{l'_1,\dots,l'_k}} \text{ with } \{l'_1,\dots,l'_k\} \subseteq \{l_1,\dots,l_k\}$$

$$\frac{G' \vDash G \qquad F' \vDash F}{G \Rightarrow F \vDash G' \Rightarrow F'} \qquad \frac{G \vDash G' \qquad F \vDash F'}{G \wedge F \vDash G' \wedge F'} \tag{5}$$

The rule $\frac{[\varphi]^{l_1,\dots,l_n}}{[\varphi]^{l'_1,\dots,l'_k}}$ is particullary usefull in the Squirrel Prover tool, as it is possible to prove a lemma for the labels l_1, l_2, l_3, l_4, and then use it for example in an equiv$^{(l_1,l_4)}(t)$ proof. We can thus prove very generic lemmas, and then use-them on specific systems.

III. Diff generalization in the Squirrel Prover

Now that there is a formal background, we can work on the generalization of the `diff` operator in the Squirrel Prover. For this, we first need to see what are the componnennts in the actual tool.

A. Overview

There are two main scopes to deal with in the Squirrel Prover :

- Protocol specification
- Lemmas and theorem proving

First in protocol specification, we formally descrbie the protocol that we will verify. This is where we use the `diff` operator, and we use keywords such as `in` and `out` to describe the exchange of messages between parties (and eventually an attacker).

Secondly, we verify properties (either trace properties or privacy properties) within lemmas or theorem. Here, we use Coq-like keywords to guide the Squirrel Prover through the resolution, with some added cryptographic tactics for dealing with hash functions, signatures etc.

However, in protocol specification we cannot use the `diff` operator for more than two systems, and lemmas / theorems expect to see systems with one or two projections, hard-coded to be `left` and `right`.

B. The issue

To see how this can be an issue, we can look at an example.

We can build a protocol that take an input (`input`), encrypt it with one secret key (`enckey`), and sign it with another secret key (`signkey`). If we want to prove that an attacker cannot obtain information about the encrypting and signing key, we need four variants :

- Two to compare encrypting with the real key, and with a random value.
- Two to compare signing with the real key, and with a random value.

Currently, we cannot write them all at once, we need to split them in an arbitrary manner :

```
process real_encrypt =
  in(cB, input);
  let crypted = enc(enckey,r,input)
  let signature = sign(crypted, diff(signkey, kS))
  out(cB, (crypted, signature))

process ideal_encrypt =
  in(cB, input);
  let crypted = enc(kfresh,r,input)
  let signature = sign(crypted, diff(signkey, kS))
  out(cB, (crypted, signature))
```

In the `real_encrypt` process, we encrypt using the real key, whereas in the `ideal_encrypt` process, we encrypt with a nonce `kfresh`, and in both, the first projection is signing with the real key, and the second projection is signing with a nonce `kS`.

This get more annoying as the process get more complicated (in reality, such process can be more than 20 lines long, so repeating them can become quite teadious. The goal would be to be able to write something like :

```
process objective =
  in(cB, input);
  let crypted = enc(diff(enckey, enckey, kfresh,
kfresh), r, input)
  let signature = sign(crypted, diff(signkey, kS,
signkey, kS))
  out(cB, (crypted, signature))
```

This make `objective` a process with 4 projections.

C. Generalization of the `diff`

The first step is to generalize the various functions in the code base to work with more than binary diffs. Some of this word were already done, as it is a wanted feature from already some time. For example, the OCaml type for the diffs in the terms can already take as many elements as wanted.

However, it is not possible to build those types with more than two elements, and most functions that use them expect bi-terms with the `left` and `right` projections.

For most functions, it is just a matter of iterating the already written logic on a list, and read the actual projections instead of assuming they will be `left / right`.

There is however one function that require more work, the `make_normal_multiterm` function. This function pushes the diff as deep as possible. For example, the term `diff(if a = b then x, if a = b then y)` become `if a = b then diff(x, y)`.

This needed generalization of all the checks made for evry possible case (every kind of terms), and made the function much more convoluted, ending with a function spanning more than 300 lines of code, not mentionning the intermediary function also generalized for the occasion.

While doing such, some sanity checks have been added to make sure that the terms given to the functions are well-formed and can fit within the logical operation, and one test started failing. The list of projections given to the function was different from the one actually in the term, and unfolding the test made us find a bug that has since been fixed.

D. Update of the parser

Finally, we needed to update the grammar and the parser of the Squirrel Prover, and here, some design decisions had to be made. We needed a new way to describe process and systems that allow more than binary systems. It would be nice to be retro-compatible with old process and system declaration (because forcing evryone to rewrite every squirrel file is not a great idea).

So, this has been done withe 2 new rules to describe process, one can now write a process by telling the arity of the term :

```
process example_process # 4 =
  in(c, input)
  out(c, diff(x, y, z, input))
```

This will create a process with 4 projections, by default named `1, 2, 3` and `4`. But if you want to explicitly name the projections, it can be done like this :

```
process example_process [first, second, third] =
  in(c, input)
  out(c, diff(x, y, input))
```

E. Result

Finally, we can now rewrite our first example within a single 4-process, like this :

```
process objective # 4 =
  in(cB, input);
   let crypted = enc(diff(enckey, enckey, kfresh,
kfresh), r, input)
  let signature = sign(crypted, diff(signkey, kS,
signkey, kS))
  out(cB, (crypted, signature))
```

IV. Conclusion

The result of this work is a more general way to write process and systems in the Squirrel Prover. This has been done in one file, and in addition to the protocol descriptions being halved, every lemma and theorem can be written only once instead of twe before, effectively dividing the file size (and proof time) by 2.

Some work can still be done to ease the life of protocol provers.

For example, when doing proofs with multi-systems, it is often needed to finish a proof with the `project` tactic (dividing a goal into as many subgoals as they are projections, and proving the goal in evry projections). It often occurs that many projections are equal, (with a 4-system, where at this point the first two are equal, and the last two are equal). In this case, it is needed to repeat the proof in every projection, but the detections of such similarities can be automated to reduce the number of added goals.

Generally, some "smartness" could be added to many tactics to work better with multi-systems.

References

[1] A.-I. Radu, T. Chothia, C. J. Newton, I. Boureanu, and L. Chen, "Practical EMV Relay Protection," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1737–1756. doi: 10.1109/SP46214.2022.9833642.

[2] D. Basin, C. Cremers, J. Dreier, and R. Sasse, "Tamarin: Verification of Large-Scale, Real World, Cryptographic Protocols," *IEEE Security and Privacy Magazine*, 2022, doi: 10.1109/msec.2022.3154689.

[3] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.*, 2001, pp. 82–96. doi: 10.1109/CSFW.2001.930138.

[4] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983, doi: 10.1109/TIT.1983.1056650.

[5] B. Blanchet, "Security Protocol Verification: Symbolic and Computational Models." [Online]. Available: https://bblanche.gitlabpages.inria.fr/publications/BlanchetETAPS12.pdf

[6] S. Goldwasser and S. Micali, "Probabilistic encryption," *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984, doi: https://doi.org/10.1016/0022-0000(84)90070-9.

[7] S. Goldwasser, S. Micali, and R. L. Rivest, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, 1988, doi: 10.1137/0217017.

[8] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An Interactive Prover for Protocol Verification in the Computational Model," in *SP 2021 - 42nd IEEE Symposium on Security and Privacy*, in Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21). May 2021. [Online]. Available: https://hal.science/hal-03172119

[9] G. Bana and H. Comon-Lundh, "Towards Unconditional Soundness: Computationally Complete Symbolic Attacker," in *Principles of Security and Trust*, P. Degano and J. D. Guttman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 189–208.

[10] G. Bana and H. Comon-Lundh, "A Computationally Complete Symbolic Attacker for Equivalence Properties," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 609–620. doi: 10.1145/2660267.2660276.