Bachelorproject

Kira Kutscher

March 24, 2019

1 Introduction

In an increasingly digitalised world it is important that we can be sure of the security of our systems. We have all heard of scandals due to breaches of security system like when Sony was hacked and almost 50.000 social security numbers stolen, or the leak of celebrity nudes from the iCloud that became widely known as "the Fappening" (both in 2014).

Both Sony and Apple are huge companies and one would expect them to have their security set up so that this kind of faux-pas cannot happen. So why did they happen?

The reason is that building proper security is hard; and proving it correct is even harder. Sometimes proofs omitted because *ain't nobody got time for that*, and when they are done it's often manually, which is dangerously error prone and leads to great trust in protocols that are in reality unsafe.

A possible way to make proofs of security more reliable and easier to conduct is to use a proof assistant. Some tools have been developed to this end, but the general problem is the difficulty of implementing proof tools that are free of bugs and inconsistencies.

1.1 Cryptography in Coq

Enter Coq, easily the most trusted proof assistant in computer science. It was first developed in 1984 [Reference, Coquand & Huet], based on the Calculus of Constructions, and since extended to the Calculus of Inductive Constructions [Reference, Christine Paulin].

Coq has been under constant development and many bugs have been fixed since it was first published. It has been used for a number of large-scale projects in mathematics and computer science like a formalised proof of the four colour theorem [Reference!].

Now, the idea should be obvious: We should use Coq in order to prove our cryptographic algorithms correct. This, however, is harder than it sounds. To see why, the reader should know a little bit about the inner workings of Coq.

The way propositions and proofs in Coq are represented is by types and terms: According to the Curry-Howard-correspondence [Reference] a logical system can be viewed as a programming language where propositions correspond to types and a proof of said proposition would take the shape of a term of said type. This representation and the requirement of reliability have introduced a few quirks into Coq which at times can make ones life hard; the ones that are important for the present development are that Coq requires all its programs to terminate with absolute certainty, and Coq does not know how to handle randomness.

So that's where the catch is: No randomness. Many cryptographic algorithms rely on randomness in order to ensure security, so how would one work with them in a non-randomised setting?

The answer is to encapsulate everything random or non-terminating in a sub-environment that can happily co-exist with Coq's termination checker. For this we use a monad, a mathematical construct which has shown to be of great use in purely functional programming. So by designing a language complete with non-termination and randomness and making its interpretation monadic, it is possible to use Coq in order to reason about it.

A number of attempts of this have been made already [Reference CertiCrypt, xhl, ALEA, FCF], and those have helped guide the present development.

The rest of this paper will be organised as follows. In Section 2 we introduce the necessary theoretical background and discuss existing approaches, after which we will move on to describe our own approach (Section 3) and state our contribution (Section 4). Lastly we will compare our development to the existing ones and present some possibilities of future work (Section 5). The paper is then rounded off with the concluding Section 6.

2 Theory and existing frameworks

2.1 $\mathcal{R}$`ml`

$\mathcal{R}$`ml` stands for random monadic language and consists of the following constructs:

$$\mathit{exp} ::= x | \mathbb{N} | \mathbb{B} | \texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2 | \texttt{let } x = e_1 \texttt{ in } e_2 | f\ e_1\ e_2\ \ldots\ e_n$$

The interpreted type of an $\mathcal{R}$`ml` expression of type τ is $(\tau \to [0,1]) \to [0,1]$, that is, a measure on the type τ. This will from now on be written as $\texttt{M}\tau$.

The interpretation of an expression e in $\mathcal{R}$`ml` is written as $[e]$, which is the measure associated with e. If e is of type $\texttt{M}\beta$ and we have a property on values of type β, Q, we can compute the probability of $[e]$ satisfying Q by giving the characteristic function $\mathbb{I}_Q$ as argument to the monadic interpretation of e.

2.2 `pwhile`

$$\mathit{exp} ::= x | \mathit{const} | \mathit{prp}\ \texttt{pred mem} | e_1\ e_2$$
$$\mathit{cmd} ::= \texttt{abort} | \texttt{skip} | x := e | x\ \$ = e | \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 | \texttt{while } b \texttt{ do } c | c_1; c_2$$

3 Our approach

4 What we contributed

5 Future work

6 Conclusion

7 Acknowledgements