DEEPSEC

DEciding Equivalence Properties in SECurity protocols

User manual

version 2.0.0

Vincent Cheval, Steve Kremer and Itsaka Rakotonirina

Université de Lorraine, Inria Nancy Grand-Est, LORIA

March 20, 2020

Contents

Introduction

The **deepsec** prover is a verification tool for cryptographic protocols. It allows the verification of security properties (expressed as a trace equivalence) of protocols described in the applied pi calculus. The tool operates in the so-called “bounded number of sessions” model: while it only allows to specify a fixed number of participants and sessions, termination is always guaranteed (though computational ressources may be exhausted if the model is too large).

Scope of this manual

This manual provides a “hands-on” introduction on how to use the tool. It will provide intuitive explanations of the language and the properties. It will also explain the different options and prov ide a reference guide for the precise syntax. It will however *not* give formal semantics nor explain the underlying algorithms. The theory underlying **deepsec** is described in (Cheval, Kremer, and Rakotonirina 2018a) and (Cheval, Kremer, and Rakotonirina 2019). Some of the implementation choices are also discussed in a tool paper (Cheval, Kremer, and Rakotonirina 2018b).

Support

Please report any bugs to `vincent.cheval@inria.fr` or file an issue on our github.

Acknowledgements

The research that led to **deepsec** was primarily supported by ERC under the EU’s H2020 research and innovation program (grant agreements No 645865-SPOOC), as well as from the French ANR projects SEQUOIA (ANR-14-CE28-0030-01) and TECAP (ANR-17-CE39-0004-01).

Download and installation

In this section we will guide you through the installation of the **DeepSec** prover and its graphical user interface **DeepSec UI**. **DeepSec** can be used independently of **DeepSec UI** but the latter requires the former to be installed. Please install both so that you can test as many feature as possible.

Installation of DeepSec

DeepSec requires **OCaml > 4.05**. It is highly recommended to install **OCaml** through `opam` instead of a native package manager, such as `apt-get` (the latest version on `apt-get` may not be the latest release of OCaml). `opam` itself may however be safely installed using your favorite package manager (see instructions for installing `opam`). To know your current version of **OCaml**, just run `ocaml --version`.

Upgrading OCaml using OPAM 1.x.x (Can be skipped if you already have ocaml 4.05 or later)

1. Run `opam switch list` (The version 4.05.0 should be displayed in the list. Otherwise run first `opam update`).
2. Run `opam switch 4.05.0` (or a more recent version).
3. Follow the instructions (at the end do not forget to set the environment by running ``eval `opam config env` ``).

Upgrading OCaml using OPAM 2.x.x (Can be skipped if you already have ocaml 4.05 or later)

1. Run `opam switch list-available` (The version `ocaml-base-compiler 4.05.0` should be displayed in the list. Otherwise, first run `opam update`).
2. Run `opam switch create 4.05.0` (or a more recent version).
3. Follow the instructions.

Installation of DeepSec from source

Deepsec requires the package **ocamlbuild** to compile which itself requires **ocamlfind**. It is important that both ocamlbuild and ocamlfind are compiled with the same version of OCaml. Running `opam install ocamlbuild` may not install ocamlfind if an instance of ocamlfind was installed on a different installation of OCaml (which sometimes happen on MacOSX). It is safer to run `opam install ocamlfind` before. Once the alpha version has been tested, we foresee to provide an `opam` package for **DeepSec** to ease the installation.

1. Run `opam install ocamlfind` (Optional if already installed)
2. Run `opam install ocamlbuild` (Optional if already installed)
3. Run `git clone https://github.com/DeepSec-prover/deepsec.git` (with a HTTPS connexion) or `git clone git@github.com:DeepSec-prover/deepsec.git` (with a SSH connexion)
4. Inside the directory `deepsec`, run `make`
5. The executable program `deepsec` has been built.

Note that two additional executables are compiled at the same time as `deepsec`: `deepsec_worker` and `deepsec_api`. The former is used by **DeepSec** to distribute the computation on multi-core architectures and clusters of computers. The latter is used to communicate with **DeepSec UI**. Both should not be used manually nor should they be moved from the `deepsec` folder.

Installation of DeepSec UI

DeepSec UI has been packaged so you don't need to compile it from the source. Just download the version according to your OS and double click. You can also directly visit DeepSec UI Releases to get the lastest version. If you need another distribution, please feel free to ask (currently no windows support…)

1. For MacOSX: deepsec-ui-1.0.0-rc3_OSX.dmg
2. For Linux:

- Debian: deepsec-ui-1.0.0-rc3_amd64.deb
- Snapshot: deepsec-ui-1.0.0-rc3_amd64.snap
- AppImage: deepsec-ui-1.0.0-rc3.AppImage

To work, **DeepSec UI** requires to know the location of the executable `deepsec_api` that was installed by **DeepSec**. When **DeepSec** will be installed through `opam` in the foreseeable future, it will be added in your `PATH` environment automatically and so **DeepSec UI** will find it automatically. Thus currently, either you can add `deepsec_api` in your `PATH` or you can manually indicate to **DeepSec UI** where it is located (in the **Settings** menu of **DeepSec UI**).

Editor modes

The **deepsec** input language is very close to the one used by the **proverif** verification tool. You may use the proverif modes (pi mode) for `emacs` and `atom` distributed at https://proverif.inria.fr.

Tutorial

The Private Authentication Protocol

We will now explain protocol verification in **deepsec** through an example. As **deepsec** specializes in verifying equivalence properties, mainly used for modelling privacy preserving properties, we will use the Private Authentication Protocol (PAP) as our example (Abadi and Fournet 2004). The protocol can be described in "Alice & Bob" notation as follows:

```
A -> B: aenc( (Na,pk(skA)), pk(skB) )
B -> A: aenc( (Na,Nb,pk(skB)), pk(skA) )     if B accepts requests from A
        aenc( Nb, pk(skB) )                  otherwise
```

Alice (A) makes a connection request to Bob (B). For this Alice sends the asymmetric encryption (`aenc`) of the pair `(Na,pk(skA)` with Bob's public key (`pk(skB)`). Here `Na` is a fresh random nonce and `pk(skA)` is Alice's public key. Here, `pk(sk)` denotes the public key corresponding to the private key `sk`. Bob may accept requests from Alice or not. The aim of the protocol is to conceal from outside observers whether Bob does accept connections from Alice or not. This is called *private* authentication. If Alice is in the list of connections accepted by Bob, Bob replies with the message `aenc( (Na,Nb,pk(skB)), pk(skB) )`, i.e. the encryption of the tuple `(Na,Nb,pk(skB))` (where `Nb` is a fresh nonce generated by Bob) with Alice's public key `pk(skA)`. Otherwise, in order to hide the connection refusal, Bob sends a decoy message `aenc( Nb, pk(skB) )`.

The modelling of the PAP protocol in **deepsec** is available in the file

`Examples/trace_equivalence/Private_authentication/PrivateAuthentication-1session.dps`

in the **deepsec** directory.

Modelling messages in deepsec

As in other symbolic models, protocol messages are modelled as *terms*. Therefore the first part of a **deepsec** file consists in the necessary declarations. To model PAP we first declare a few constants.

```
free ca, cb, c.
free ska, skb, skc [private].
```

Here, `ca, cb, c` are so-called *free* names: they model public constants, that are known to the adversary. In PAP `ca, cb, c` will be channel names, as we will see below. On the other hand we need to declare *secret* keys. For this we use *private* names `ska, skb, skc` that are declared with the additional attribute `[private]`.

Next, we need to declare function symbols to represent asymmetric encryption.

```
fun aenc/2.
fun pk/1.
```

The function symbol `aenc` is declared to be of arity 2 using the notation `/2`. Public keys are of arity 1, as they are intended to take a secret key as argument.

> **Note:** public names *vs* function symbols of arity 0
>
> It is possible to declare function symbols of arity 0, e.g. write `fun c/0`. This is equivalent to declaring a free name `free c`.

> **Note:** alternate modelling of secret keys using private function symbols
>
> In the modelling proposed above, we intend to compute the public key by applying the function `pk` to the secret key, e.g. `pk(ska)` would be A's public key. An alternate way of modelling can be to derive both the public and secret key from an identity: we could declare a *private* function symbol `fun sk/1 [private].`. Then, `pk(a)` and `sk(a)` represent A's public, respectively private, key. Declaring the function symbol `sk` as private implies that the attacker cannot apply this function symbol.

Currently, we have declared function symbols `aenc` and `pk`, but nothing indicates that these functions represent asymmetric encryption. We will use *rewrite rules* to give meaning to these function symbols.

```
reduc adec(aenc(x,pk(y)),y) -> x.
```

This rule indicates that an attacker can apply decryption `adec`; if the keys match (which is required as we use the same variable `y` as arguments in encryption and decryption) then the result of applying decryption returns the plaintext `x`.

> **NOTE**: constructor-destructor algebras
>
> You may have noticed that we did not declare the `adec` symbol. This is because `adec` is a *destructor*, while declared function symbols are *constructors*. Destructors may actually not occur in protocol messages: if the above rewrite rule does not succeed the evaluation will *fail*. For example, the evaluation of the terms `adec(aenc(m,pk(ska)),skb)` and `adec(c,ska)` would both fail.

> **NOTE**: deterministic vs randomized encryption
>
> Example why sometimes randomized encryption is needed.

Modelling protocols as processes

To complete.

Alice's role

```
let processA(ca,ska,pkb) =
  new na;
  out(ca,aenc((na,pk(ska)),pkb));
  in(ca,x).
```

Bob's role

```
let processB(cb,skb,pka) =
  in(cb,yb);
  new nb;
  let (yna,=pka) = adec(yb,skb) in
    out(cb,aenc((yna,nb,pk(skb)),pka))
  else out(cb,aenc(nb,pk(skb))).
```

and the main process

```
let ProcessAB =
  out(c,pk(ska));
```

```
    out(c,pk(skb));
    out(c,pk(skc));
    (
      processA(ca,ska,pk(skb)) | processB(cb,skb,pk(ska))
    ).
```

Verifying anonymity

Add a “second” main process

```
  let ProcessCB =
    out(c,pk(ska));
    out(c,pk(skb));
    out(c,pk(skc));
    (
      processA(ca,skc,pk(skb)) | processB(cb,skb,pk(skc))
    ).
```

and add query

```
  query trace_equiv(ProcessAB,ProcessCB).
```

To verify use command

```
  $ deepsec PrivateAuthentication-1session.dps
```

Deepsec says trace equivalent.

What happens when we remove the decoy message? Attack found.

Distributing the computation

Big scenario increases computation time. Distribute the computation.

The deepsec User Interface

Using the GUI in the example.

Input grammar

We describe in details the input grammar of **DeepSec**.

Type system

Function symbols

Processes

Precedences

Queries

Command-line options

Command-line options of DeepSec.

References

[AF04] Martín Abadi and Cédric Fournet. “Private authentication”. In: *Theor. Comput. Sci.* 322.3 (2004), pp. 427–476.

[CKR18a] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. “DEEPSEC: Deciding Equivalence Properties in Security Protocols - Theory and Practice”. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*. Accepted for publication. San Francisco, CA, USA: IEEE Computer Society Press, May 2018.

[CKR18b] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. “The DEEPSEC prover”. In: *Proceedings of the 30th International Conference on Computer Aided Verification, Part II (CAV'18)*. Oxford, UK: Springer, July 2018. DOI: 10.1007/978-3-319-96142-2_4. URL: https://hal.inria.fr/hal-01763138/document.

[CKR19] Vincent Cheval, Steve Kremer, and Itsaka Rakotonirina. “Exploiting Symmetries When Proving Equivalence Properties for Security Protocols”. In: *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS'19)*. London, UK: ACM, Nov. 2019, pp. 905–922. DOI: 10.1145/3319535.3354260.