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

April 8, 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

A dedicated package for syntax highlighting, `language-deepsec`, is available for installation within `atom`. Given that the **deepsec** input language is very close to the one used by the **proverif** verification tool you may use the proverif mode (proverif-pi-mode) for `emacs` distributed with proverif.

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/tutorial/pap-1-session.dps
```

in the **deepsec** directory. We suggest that you move to that directory and make sure that the **deepsec** executable is in your path.

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

Here, `c` is a so-called *free* name: free names model public constants, that are known to the adversary. In PAP `c` will be a channel name, 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`, or `const c`. 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
>
> You may also note that we modelled encryption as a *deterministic* function. Of course, a secure encryption scheme needs to be randomized, but in this particular example this simplified version is sufficient. This means in particular that the attacker can distinguish messages `aenc(0,pk(ska))` and `aenc(1,pk(ska))` where `0` and `1` are constants as he could simply re-encrypt these constants (supposing he knows the public key). It is however easy to model asymmetric encryption by adding a random element, making `aenc` a ternary function.

> **Note**: multiple rewrite rules for a single destructor.
>
> Note that if a destructor function require several rewrite rules, they should be defined inside the same reduc.
>
> ```
> reduc
>    exists_double(x,x,y) -> ok;
>    exists_double(x,y,x) -> ok;
>    exists_double(y,x,x) -> ok.
> ```

Modelling protocols as processes

We now need to model the behaviour of Alice and Bob. One can think of a protocol as a distributed program. Each local program of this system will be represented by a *process*. We can model Alice's role by the following process `processA`.

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

The process has 2 arguments: `ska` is the secret key of the agent running this process, and `pkb` is the public key of the agent to whom we want to connect. First, the process generates a fresh random nonce using the command `new na`. Next, it sends on channel `c` the encryption of the pair `(na,pk(ska))` encrypted with the recipient's public key `pkb`, as dictated by the protocol. Finally, the process expects an input, modelled as `in(c,x)`. Normally, one would expect additional processing of the input message, which we omit here for simplification.

> **Note** private names vs new names
>
> In the above example we use `new na` to create a fresh, private name na. This is again equivalent to declaring a free, private name, as we did for `ska, skb` and `skb`. However, the `new` construct is useful when a different, fresh name should be created in every instance of the process: if we execute several instances of `processA` a distinct fresh name `na` is created in each copy.

Next, we model Bob's behaviour by the process `processB`.

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

This process introduces several new constructs that require explanations. The first action of the process is to input a message on channel `c` through the instruction `in(c,yb)`. As a consequence the message that is received will be bound to the variable `yb`. While the expected message is `aenc((na,pka)),pk(skb))` we need to keep in mind that this message may actually be provided by the attacker and may be an arbitrary message the attacker is able to forge. Therefore we need to *parse* the message and perform a number of tests. All of this is done here in a condensed form using a `let` instruction. We first decrypt (apply `adec`) the received message (referred to by the variable `yb`) with the secret key `skb`. Note that if decryption fails, we will enter the `else` branch of the `let` instruction. Next, we check that the results is a pair: the first element of the pair is bound to the variable `yna` and we check that the second variable of the pair equals the public key `pka`, i.e., the public key of a person we accept connections from. An expanded form could be written as follows.

```
let yplain = adec(yb,skb) in
   let (yna,ypka) = yplain in
      if ypka = pka then
         out(cb,aenc((yna,nb,pk(skb)),pka))
      else out(cb,aenc(nb,pk(skb)));
   else out(cb,aenc(nb,pk(skb)));
else out(cb,aenc(nb,pk(skb))).
```

This form is however rather lengthy and requires duplicating else branches, which is why the above syntactic sugar is often convenient.

> **Note:** tuples in **deepsec**
>
> We have seen in the above example that we used notations `(a,b)` and `(a,b,c)` for tuples without explicitly declaring function symbols for pairs and triples. Actually, **deepsec** has built-in support for tuples. For each tuple of arity *n* occurring in the processes **deepsec** will define the constructor `(_, ... ,_)` of arity n and corresponding destructors `reduc proj_i_n (x1, ... ,xi, ..., xn) = xi` (for all $1 \leq i \leq n$). These destructors are used implicitly in the let instruction for projecting the elements of the tuple.

Finally, we put all the pieces together ia main process `ProcessAB` modelling the entire system.

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

The system first outputs the public keys, so that they become known to the attacker. Then the system indicates that processes `processA` and `processB` are executed in parallel (each with its parameters).

Verifying private authentication

We are now interested in modelling anonymity. Anonymity is generally modelled as the indistinguishability of two systems. We therefore define a second system `ProcessCB`.

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

The difference with previous system `ProcessAB` is the parameter `skc`, rather than `ska`, given to `processA` and `processB`. Hence, `ProcessAB` models the situation where B is willing to receive connections only from A, while in `ProcessCB`, B accepts connections only from C. The goal of private authentication is to hide from whom connections are accepted. Indistinguishability can be modelled by trace equivalence. We can therefore query **deepsec** to check trace equivalence between these two systems.

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

To verify this query we use the command

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

deepsec will indeed confirm that this kind of anonymity is satisfied by outputting (among some other messages) that

```
Result query 1: The two processes are trace equivalent.
```

Looking at the protocol this is intuitively due to the decoy message sent in the `else` branch of `processB`. What happens when we remove the decoy message? For this we simply replace the else branch with `else 0` (or omit it completely), see the file `PrivateAuthentication-1session-attack.dps`. We can now run **deepsec** on this modified file.

```
$ deepsec pap-1-session-attack.dps
```

This time, **deepsec** will report an attack:

```
Result query 1: The two processes are not trace equivalent.
```

Indeed, when the attacker sends the message `aenc((n,pk(ska)),pk(skb))` to B, only the first system will send a reply.

> **Note:** multiple input files
>
> **deepsec** can take several files as input. For example you may run
>
> ```
> deepsec pap-1-session.dps pap-1-session-attack.dps
> ```

More complex scenarios and scaling up

In the previous section we considered a very simple scenario and our verification checked whether private authentication holds when we have one instance of `A` and `B`. Often, protocols may be secure when considering a single session, but attacks may arise when multiple sessions are executed in parallel.

Let us see what happens when we consider two instances of each role resulting into the following declarations.

```
let ProcessAB =
  out(c,pk(ska));
  out(c,pk(skb));
  out(c,pk(skc));
  (
    processA(ska,pk(skb)) | processB(skb,pk(ska)) | // B expects to talk to A
    processA(ska,pk(skb)) | processB(skb,pk(ska))   // B expects to talk to A
  ).

let ProcessCB =
  out(c,pk(ska));
  out(c,pk(skb));
  out(c,pk(skc));
  (
    processA(skc,pk(skb)) | processB(skb,pk(skc)) | // B expects to talk to C
    processA(ska,pk(skb)) | processB(skb,pk(ska))   // B expects to talk to A
  ).
```

> Note: bounded replication
>
> When considering multiple sessions it is common to put in parallel several identical instances. For example, the process `ProcessAB` duplicates `processA(ska,pk(skb))` and

> `processB(skb,pk(ska))`. In more complex scenarios we may want to copy more processes a large number of times. Therefore **deepsec** provides a convenient operator `!^n`: `!^n P` is syntactic sugar for `n` parallel copies of *P* where `n` is a positive integer. In the above example
>
> ```
>    processA(ska,pk(skb)) | processB(skb,pk(ska)) |
>    processA(ska,pk(skb)) | processB(skb,pk(ska))
> ```
>
> could have been replaced by
>
> ```
>    !^2 processA(ska,pk(skb)) | !^2 processB(skb,pk(ska))
> ```

We can run

```
  $ deepsec pap-2-session.dps
```

and observe that still no attack is found. However, the verification time increases: while the result is instantaneous for 1 session it now takes several seconds on a standard laptop. This is due to the fact that **deepsec** has to explore **all** possible interleavings, whose number is exponential.

While the verification time is still moderate for 2 sessions this is not the case anymore when we add a third session.

```
  $ deepsec pap-3-session.dps
```

will take *much* longer. How can we ensure that the protocol cannot be attacked with 3 sessions, or more?

Distributing the computation

A first way to scale up is to distribute the computation. By default, **deepsec** checks how many physical cores your machine has and distributes the computation on these cores by creating the same amount of *workers*. To activate the distributed computation with a different number of workers, `deepsec` should be run with the option `-l n` (or `--local_workers n`) where `n` is the number of desired local workers.

It is also possible to distribute computation on several machines. To do so, deepsec requires an ssh connexion between the localhost and the distant machine, using ssh key authentication, so that no password is required. The computation on a distant machine is configured with the command line option `-w <host> <path> <n>` (or `--distant_workers <host> <path> <n>`). The parameter `<host>` is the ssh login and address (e.g my_login@my_host). The parameter `<path>` should indicate the path to the deepsec directory on the distant machine. Finally, the parameter `<n>` represents the number of cores that should be dedicated by this distant machine to the computation of the input file.

Note that the option `-distant_workers` must be used for each distant machine.

```
deepsec -w login1@host1 tools/deepsec 15 \
        -w login2@host2 deepsec auto my_file.dps
```

In this command line, the first machine should be accessible with `ssh login1@host1` and the **deepsec** directory should be located at `~/tools/deepsec` on this machine. Similarly, the second machine should be accessible with `ssh login2@host1` and the **deepsec** directory should be located at `~/deepsec`. If the connexions to both machines are successful, **deepsec** will distribute the computation between the local and the 2 distant machines: 15 cores are used on the first machine and, by specifying `auto`, all available physical cores on the second machine.

> ***Important***: The localhost and distant machines must have exactly the same version of **deepsec** (the Git hash is displayed when running `deepsec` without parameters or with the option `--help`), compiled with the same version of **OCaml**.

Session equivalence

Distribution of the computation may gain a constant speed-up factor: going from a 20 hours computation to a 1 hour computation is indeed much appreciated, but may not solve the more fundamental problem of the exponential blowup.

This is why **deepsec** proposes another, more efficient proof technique. The underlying idea is to prove a stronger equivalence relation, that we call *equivalence by session*. This equivalence significantly decreases the number of interleavings and exploits the structure of the processes. Often, we want to show the equivalence of processes that are of the form

```
let P = P1 | ... | Pn
let Q = Q1 | ... | Qn
```

The rough idea of equivalence by sessions is to match parallel sessions rather than individual actions. Here, for instance, one may try to match all actions of `P1` by all actions of say `Q3`, all actions of `P2` by all actions of `Q1`, etc. **deepsec** still needs to explore all possible matches, but this equivalence allows many more optimizations than the initial trace equivalence.

For example, we may try to verify a complex scenario with 5 sessions on PAP.

```
$ deepsec pap-session-equiv-5-sessions.dps
```

Now this computation terminates in about a minute on a standard laptop (with two cores). Distributing this computation could of course improve the verification time.

> **Note:** Equivalence by session and false attacks
>
> Why shouldn't one always use the more efficient equivalence by session ? As explained above equivalence by session is a stronger equivalence than trace equivalence. Therefore whenever equivalence by session is satisfied, trace equivalence also holds. The converse may however not be true, and equivalence by session may lead to a *false attack* (with respect to trace equivalence). This is witnessed by the following small example
>
> ```
> let P = out(c,a) ; out(c,a).
> let Q = out(c,a) | out(c,a).
> query trace_equiv(P,Q).
> query session_equiv(P,Q).
> ```
>
> and can be tested using **deepsec**.
>
> ```
> $ deepsec trace-vs-session.dps
> ```

> **Note:** Syntactic restriction
>
> The theory of equivalence by session requires that all channels are only (public or private) names or constants, i.e., no complex terms, nor variables.

Partial-order reduction techniques

Probably the most effective way to fight state explosion are partial order reduction (POR) techniques. **Deepsec** implements powerful POR optimizations, that were designed in (Baelde, Delaune, and Hirschi 2015). These techniques are however only sound on a class of *action-determinate* processes: a process is action determinate when it never can reach a state where two outputs, or two inputs on a same channel are executable. Moreover, the processes may not use *private* channels. A simple, sufficient criterion is to check that syntactically no outputs on a same channel appear in parallel, and similarly for inputs, and that all channels are public. **Deepsec** automatically checks this criterion, and when satisfied enables POR techniques.

A pragmatic way to ensure action determinacy is to use a different channel name for each process in parallel. It is easy to modify the specification of PAP in that way. This modelling allows for a spectacular efficiency gain. The verification of a scenario with 9 sessions terminates in a few seconds.

```
$ deepsec pap-por-9-sessions.dps
```

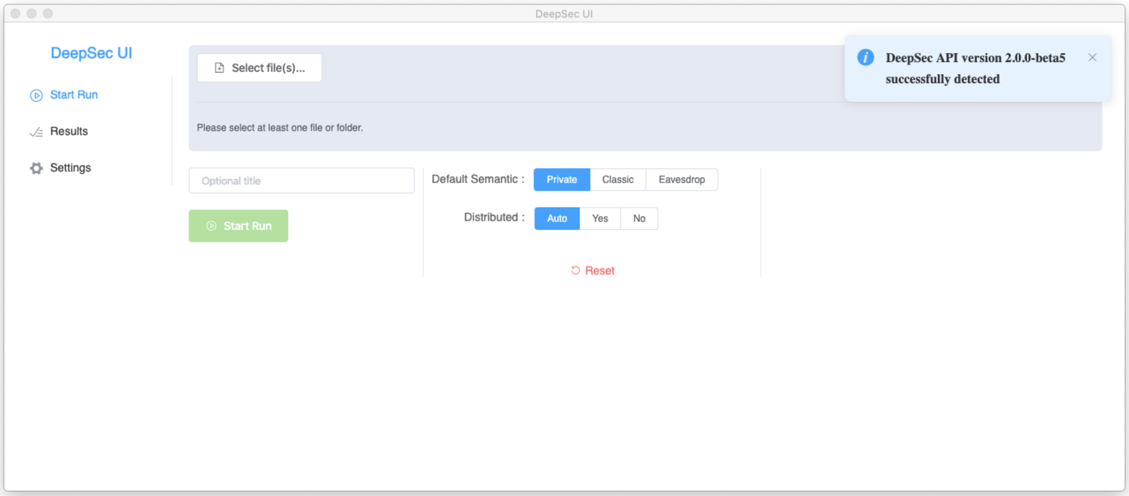
Again, one may wonder why one should not always use different channel names for parallel processes? Intuitively, using different channels for parallel sessions allows the attacker to identify the session that has sent the message. While this works well for the PAP protocol, some protocols precisely rely on this *sender ambiguity* to ensure some form of anonymity.

The deepsec User Interface

Deepsec also comes with a graphical user interface (GUI). The GUI is intended to provide an easy to use environment for using **deepsec**, browsing the results and simulating attacks as well as equivalence proofs.

The GUI is launched by executing the **DeepSec UI** application. This is a standalone application that communicates with **deepsec** by making calls to the **deepsec_api** executable. Therefore, you should make sure that this executable is in your system path. (Otherwise we can manually configure the path to **deepsec_api**, see below).

When launching **DeepSec UI** you should arrive at the following welcome screen.



DeepSec API version 2.0.0-beta5 successfully detected

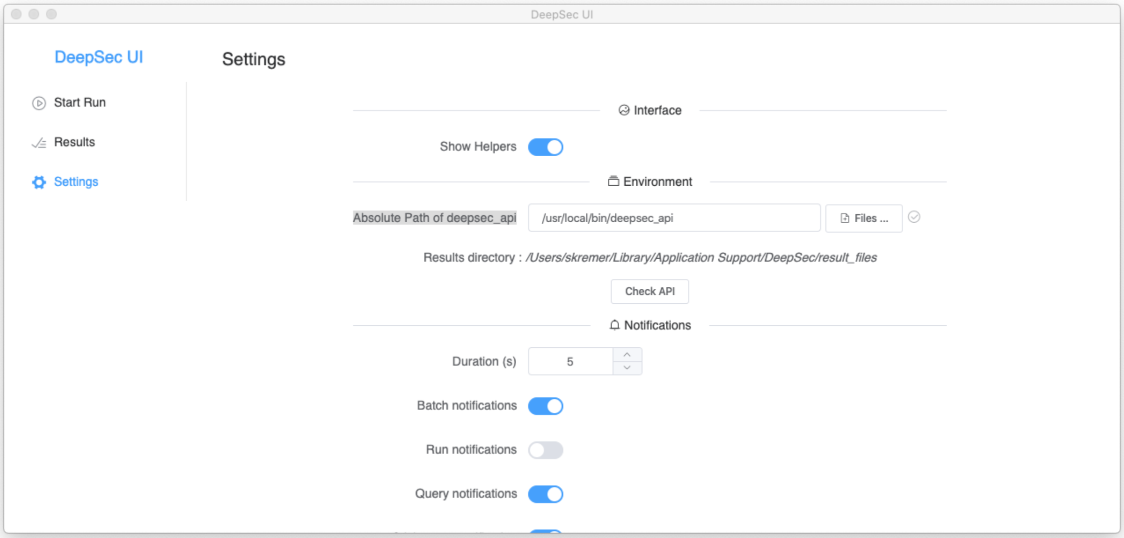
Please select at least one file or folder.

Default Semantic : Private

Distributed : Auto

From this screen you can navigate through the 3 main sections of the GUI (displayed on the left): * Start Run * Results * Settings

Normally you should see a “pop-up” confirming that **deepsec_api** has been successfully detected. If not you will get a *warning* pop-up. This notification only appears for a few second. If the executable was not detected you may manually specify the path by clicking on *Settings* and providing the *Absolute Path of deepsec_api*.



Settings

Interface

Show Helpers ●

Environment

Absolute Path of deepsec_api: /usr/local/bin/deepsec_api

Results directory : /Users/skremer/Library/Application Support/DeepSec/result_files

Notifications

Duration (s): 5

Batch notifications ●

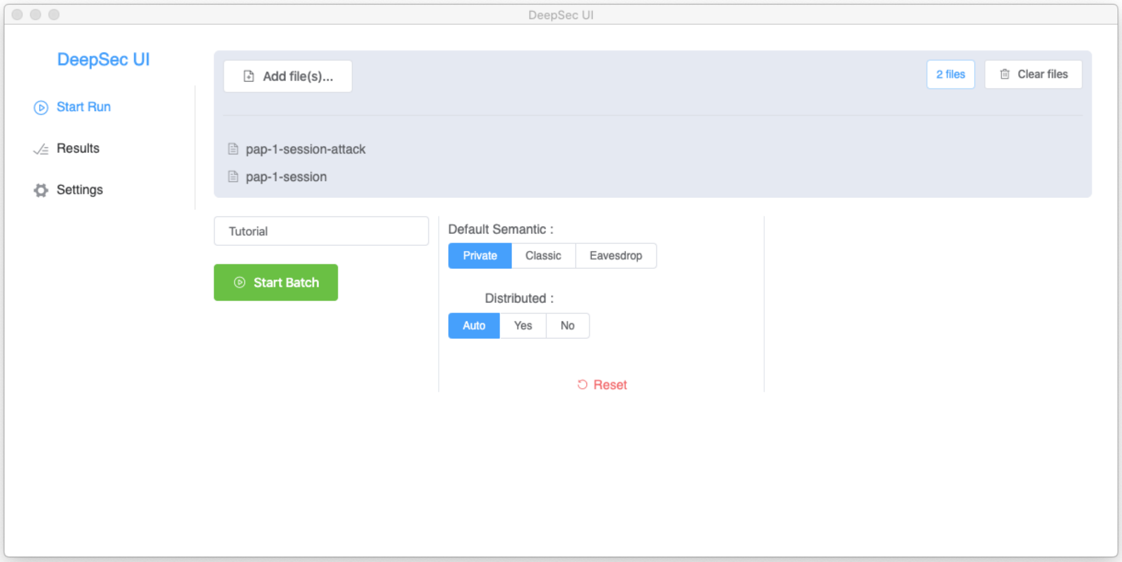
Run notifications ○

Query notifications ●

You may test that the executable is indeed available in the specified path by clicking on *Check API*.

We can now navigate back to the *Start Run* section and select the files with the specifications to be analysed. Let us select the `pap-1-session` and `pap-1-session-attack` that we used previously (available in `Examples/tutorial/` in the **deepsec** folder). The GUI allows you to select multiple files and such a collection of files is called a *batch*. Each file of this *batch* is called a *run* and such a run may contain multiple *queries*, as several queries may be specified in a same file.

In order to reference this batch we may provide a title, e.g. `Tutorial`. The `Start Run` section also allows for more advanced settings (*Semantics*, *Distributed*), but we currently keep the default settings.



pap-1-session-attack

pap-1-session

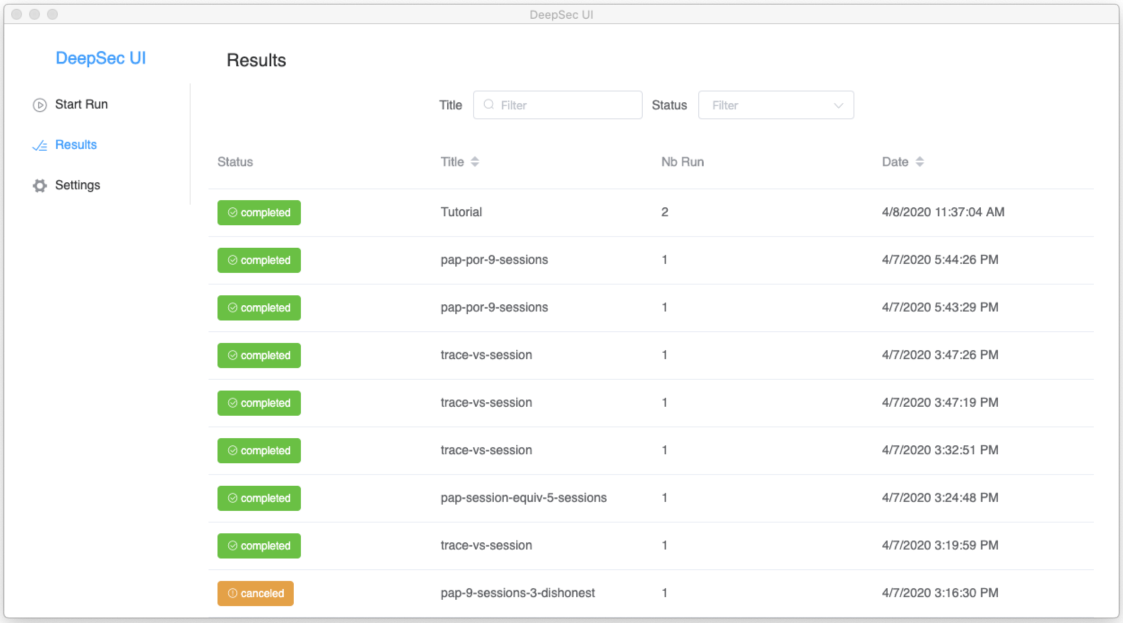
Tutorial

Default Semantic : Private

Distributed : Auto

We can now start the verification by clicking on the *Start Batch* button. Pop-up windows will notify about the status of the verification.

Navigating to the results section we now see the list of all previous verifications including the *Tutorial* batch.

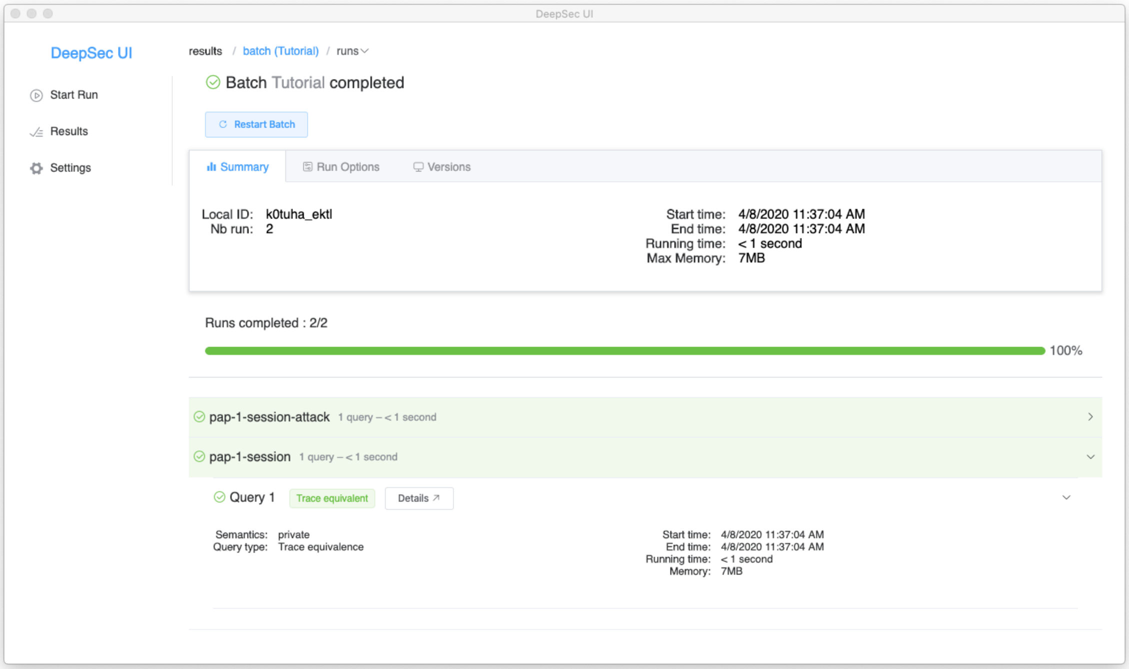


Results

Status	Title	Nb Run	Date
completed	Tutorial	2	4/8/2020 11:37:04 AM
completed	pap-por-9-sessions	1	4/7/2020 5:44:26 PM
completed	pap-por-9-sessions	1	4/7/2020 5:43:29 PM
completed	trace-vs-session	1	4/7/2020 3:47:26 PM
completed	trace-vs-session	1	4/7/2020 3:47:19 PM
completed	trace-vs-session	1	4/7/2020 3:32:51 PM
completed	pap-session-equiv-5-sessions	1	4/7/2020 3:24:48 PM
completed	trace-vs-session	1	4/7/2020 3:19:59 PM
canceled	pap-9-sessions-3-dishonest	1	4/7/2020 3:16:30 PM

Clicking on the tutorial batch we can display additional information. You may inspect the Run options and Versions to see the precise parameters and software versions used to run this batch.

Clicking on `pap-1-session` and then `Query 1` allows to reveal additional information about the individual run and query.



Batch Tutorial completed

Local ID: k0tuha_ektl
Nb run: 2

Start time: 4/8/2020 11:37:04 AM
End time: 4/8/2020 11:37:04 AM
Running time: < 1 second
Max Memory: 7MB

Runs completed : 2/2

100%

pap-1-session-attack 1 query – < 1 second

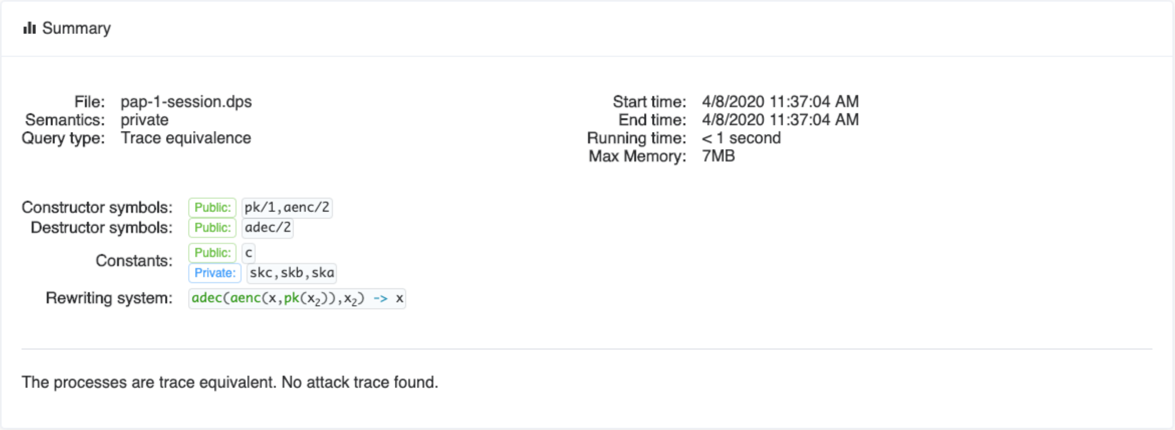
pap-1-session 1 query – < 1 second

Query 1

Semantics: private
Query type: Trace equivalence

Start time: 4/8/2020 11:37:04 AM
End time: 4/8/2020 11:37:04 AM
Running time: < 1 second
Memory: 7MB

We can now inspect the *Details* of `Query 1`. The first part of the screen provides a summary of the query, recalling the declarations of the file and showing the result of the verification.



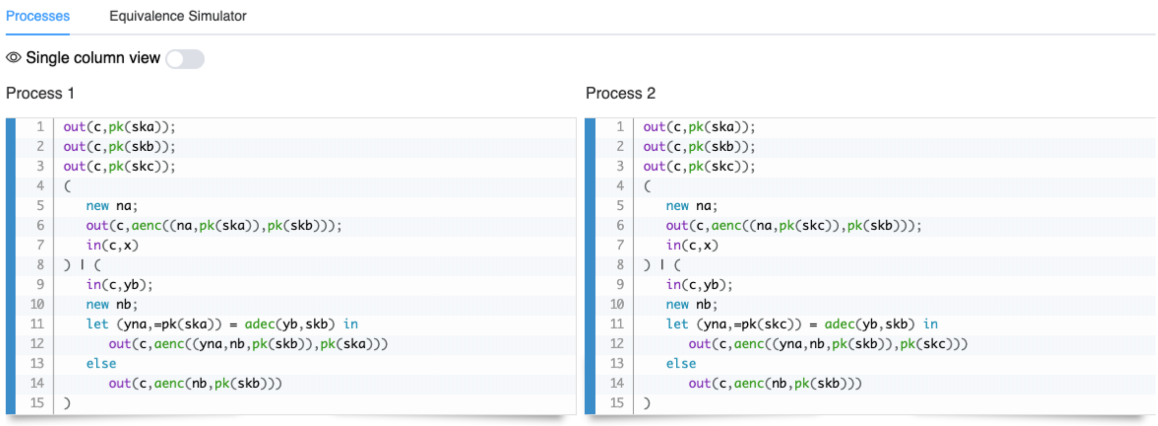
File: pap-1-session.dps
Semantics: private
Query type: Trace equivalence

Start time: 4/8/2020 11:37:04 AM
End time: 4/8/2020 11:37:04 AM
Running time: < 1 second
Max Memory: 7MB

Constructor symbols: Public: pk/1,aenc/2
Destructor symbols: Public: adec/2
Constants: Public: c
Private: skc,skb,ska
Rewriting system: adec(aenc(x,pk(x_2)),x_2) -> x

The processes are trace equivalent. No attack trace found.

The second part shows the two processes of the query.



Process 1

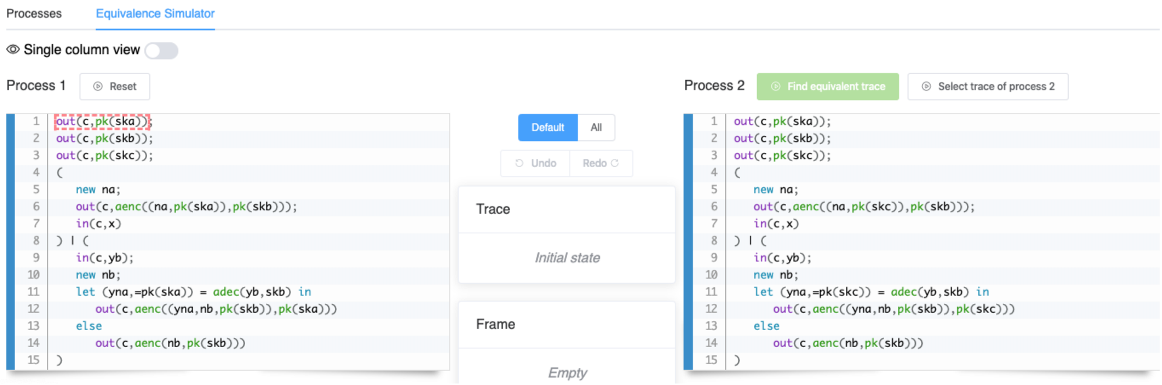
```
out(c,pk(ska));
out(c,pk(skb));
out(c,pk(skc));
(
   new na;
   out(c,aenc((na,pk(ska)),pk(skb)));
   in(c,x)
) | (
   in(c,yb);
   new nb;
   let (yna,=pk(ska)) = adec(yb,skb) in
      out(c,aenc((yna,nb,pk(skb)),pk(ska)))
   else
      out(c,aenc(nb,pk(skb)))
)
```

Process 2

```
out(c,pk(ska));
out(c,pk(skb));
out(c,pk(skc));
(
   new na;
   out(c,aenc((na,pk(skc)),pk(skb)));
   in(c,x)
) | (
   in(c,yb);
   new nb;
   let (yna,=pk(skc)) = adec(yb,skb) in
      out(c,aenc((yna,nb,pk(skb)),pk(skc)))
   else
      out(c,aenc(nb,pk(skb)))
)
```

We may note that all processes defined by a `let ... = ...` construct have been inlined. An interesting functionality here is the *Equivalence Simulator* (click above the processes): it allows to define a trace on one process and have **deepsec** find the equivalent trace on the other process. “Playing around” with the processes should allow the user to understand *why* the two processes are equivalent.

For instance, we could select a trace on Process 1. Available choices are highlighted and the user may select one of them.



Process 1

```
out(c,pk(ska));
out(c,pk(skb));
out(c,pk(skc));
(
   new na;
   out(c,aenc((na,pk(ska)),pk(skb)));
   in(c,x)
) | (
   in(c,yb);
   new nb;
   let (yna,=pk(ska)) = adec(yb,skb) in
      out(c,aenc((yna,nb,pk(skb)),pk(ska)))
   else
      out(c,aenc(nb,pk(skb)))
)
```

Trace

Initial state

Frame

Empty

Process 2

```
out(c,pk(ska));
out(c,pk(skb));
out(c,pk(skc));
(
   new na;
   out(c,aenc((na,pk(skc)),pk(skb)));
   in(c,x)
) | (
   in(c,yb);
   new nb;
   let (yna,=pk(skc)) = adec(yb,skb) in
      out(c,aenc((yna,nb,pk(skb)),pk(skc)))
   else
      out(c,aenc(nb,pk(skb)))
)
```

Initially, a single action `out(c,pk(ska));` is available and there is no other choice than select this one. This is a direct communication with the attacker on a channel; hence, the attacker needs to provide a *recipe* to compute this channel name. As here it is a constant `c` the tool proposes the recipe and we can simply validate. The three first actions do not offer any choice, and we can select the trace

`out(c,ax1) · out(c,ax2) · out(c,ax3)`

resulting into the *frame*

`ax1 -> pk(ska) · ax2 -> pk(skb) · ax3 -> pk(skc)`

Next we have the choice between two actions `out(c,aenc((na,pk(ska)),pk(skb)))` and `in(c,yb)`. Let us first select the output and then the input `in(c,yb)`. Here we need to provide the *recipe* for the value to input (click on the “pencil”). Let’s say we want simply to forward the last output – the recipe for this is the last element of the frame and we may indicate `ax_4`.

We see that this input satisfies the test in the `let` construct. Now we can select the output `out(c,aenc((yna,nb,pk(skb)),pk(ska)))` and require the tool to *Find equivalent trace* (above Process 2).

We can now “walk through” the equivalent trace using the `<Prev` and `Next>` buttons. We see that this results into a statically equivalent frame.

You may also try to see what happens if you use a different recipe for `in(c,yb)`:

```
aenc((#n,ax_1),ax_2)
```

In this recipe the attacker encrypts himself a fresh name `#n` of his own (fresh names are prefixed by `#`) and `pk(a)` (specified by `ax_1`) with the `pk(c)` (specified by `ax_3`). While this results in the same message as `ax_4` `aenc((yna,nb,pk(skb)),pk(ska))` in process 1 this is not the case in process 2. Indeed, when requesting to vind an equivalent trace we see that process 2 will move into the `else` branch and send the decoy message `aenc(nb,pk(skb))`. Nevertheless, the two resulting frames are statically equivalent.

Advanced options

Deepsec provides a number of *advanced* options which require a bit more understanding of the semantics and inner working of **deepsec**. Below we give a slightly more in-depth explanation and pointers to relevant papers.

Partial Order Reductions

Partial Order Reductions (POR) are a technique to battle state explosion. Given that the number of interleavings of parallel processes is exponential, POR techniques try to avoid the need to explore *all* interleavings. For example, given actions a, b, c, d, sometimes the interleavings *abcd* and *acbd* may be completely equivalent in the sense that neither of them increases the attacker's power to distinguish. One may think of this as a *partial order*: we need to consider all interleavings such that $a < b, c < d$ where < is the precedence relation, but b and c are not ordered. POR techniques then only explore one representative of each class of equivalent interleavings.

Deepsec implements the POR techniques presented in (Baelde, Delaune, and Hirschi 2015). The techniques are correct for a class of *action determinate* processes. *Deepsec* checks a slightly stronger, syntactic condition: there are no two parallel processes that have two outputs, respectively inputs, on the same channel, and all channels are public.

By default, the POR optimization is activated automatically whenever this syntactic condition is satisfied. One can manually disable the POR optimization, using either the command line or the graphical user interface. Note that even if POR is set manually to true, it does not change the behavior of non determinate processes in order to guarantee soundness of the result.

Choosing the semantics

The applied pi calculus comes with formal semantics under the form of a *transition relation* between processes (Abadi, Blanchet, and Fournet 2018). Minor, apparently insignificant variants have been considered in the literature. In (Babel, Cheval, and Kremer 2020), typical small variants of the semantics of public communications, occurring in the literature, are studied and 3 different semantics are considered:

- the classic semantics,
- the private semantics, and the
- eavesdrop semantics.

We will explain the difference between these semantics on an example.

```
let P = out(c,t).P1 | in(c,x).P2
```

We suppose that `c` is a public name, i.e., the channel is known to the adversary.

In the *classical semantics*, as defined in the original paper presenting the applied pi calculus (Abadi, Blanchet, and Fournet 2018), the process `P` may reduce in different ways:

- `P` may perform an *internal communication* and reduce to `P1 | P2{t/x}` that is the two parallel processes perform the output and the corresponding input and the variable `x` is replaced by the term `t` in process `P2`; an important point is that this is an *internal* action and therefore not visible to the adversary;

- `P` may perform a visible output on channel `c` and continue as `P1 | in(c,x).P2`, adding `t` to the attacker knowledge;
- `P` may perform a visible input on channel `c` and continue as `out(c,t).P1 | P2{u/x}` where `u` is a term provided by the attacker.

In the *private semantics* we remove the possibility of internal communication on a public channel. Internal communications are only possible on private channels. These semantics have been considered in many papers, because an adversary can explicitly forward the term `t` by successively performing a visible output and a visible input with term `t` (which can be provided by the attacker as it was added to his knowledge just before). An advantage of these semantics is that verification is more efficient as less interleavings need to be considered.

While the two semantics are equivalent for reachability properties, they happen to be incomparable, in general, when verifying trace equivalence (or other equivalences), as shown in (Babel, Cheval, and Kremer 2020): two processes may be trace equivalent in the private semantics, but not in the classical semantics, and vice-versa.

In the *eavesdrop semantics*, all three possible reductions of the classical semantics are considered. However, when performing an internal communication on a public channel, this action becomes visible to the attacker and the term is added to his knowledge.

Whenever trace equivalence holds in the eavesdrop semantics it also holds in both the classical and private semantics. Therefore, verification with the eavesdrop semantics is a conservative choice.

It was also shown in (Babel, Cheval, and Kremer 2020) that trace equivalence coincides on all three semantics for the class of strongly determinate processes, that is the class on which **deepsec** enables POR techniques. The default semantics of **deepsec** are the private semantics which yield more efficient verification. The semantics can be modified using either the command line or the graphical user interface.

Distributing the computation

Deepsec allows to distribute the computation on multiple cores, as well as on multiple servers. To explain the distribution we need to give a high-level overview on how **deepsec** verifies trace equivalence.

To check trace equivalence **deepsec** needs to compute a large symbolic execution tree, called the *partition tree*: each path in this tree corresponds to a symbolic trace, i.e. a trace that may contain non instantiated variables. Each node in the tree regroups the set of equivalent, symbolic processes, after the execution of the symbolic trace leading to this node. Checking trace equivalence between processes `P` and `Q` then consists in verifying that each node contains at least one process derived from `P` and one derived from `Q`.

The main idea of the distribution is to let different cores explore different branches of the tree.

- The first step is the *job creation*: we generate, in a breadth-first manner, a number of nodes whose subtrees need to be generated and explored. These nodes are stored in a queue and called *jobs*. The number of such jobs that are initially generated
- Next each *worker*, i.e., each core, fetches a job and verifies the underlying subtree. When a job is finished, the *worker* fetches the next job in the queue.

- As the tree is not balanced, some jobs may require much more computation than others (and we cannot predict wich jobs are taking more time). Therefore, some workers may become idle while others still have a large subtree to verify. When a worker becomes idle, i.e., the queue of pending jobs is empty, we start a *timer*: after the time-out we kill ongoing jobs, and start a new job creation phase to distribute the computation of these remaining large subtrees. This is called a new *round*.

The time-out before staring the next round avoids killing on-going work that is about to finish and get into a job creation – kill worker loop.

The command line and graphical user interface allow to set the values for

- the number of local and remote workers – when set to 'auto' all available, physical cores are used;
- the minimum number of jobs created in the job creation phase – when set to 'auto' the number of jobs is 100 × the total number of workers;
- the round timer – the default value is 120 seconds.

The graphical User Interface

Deepsec comes with a Graphical User Interface (GUI). **DeepSecUI** is a standalone application that interacts with the `deepsec_api` executable.

When launching the application you should see a pop-up message stating *"DeepSec API version x.y.z successfully detected"*. This means that **DeepSecUI** was able to successfully locate the `deepsec_api` executable. If you see a message *"DeepSec API path is not set"* you must either add the path to your executable in your system PATH, or manually indicate *Absolute Path of deepsec_api* in the *Settings*.

The GUI should be rather intuitive and mostly self-explanatory if you are familiar with protocol verification. We document each of the sections of the GUI and its main items.

Start run

DeepSecUI allows to select several files and run **deepsec** on these files for verification. Such a set of files is called a *batch*. Running **deepsec** on a single file is called a *run*. Each file may contain several *queries* to be verified.

The *Select files* button allows to select one or more files for a run, respectively batch. It is possible to specify a title for a run or batch. If specified, this title will be used in the *Results* section to refer to the batch, or run.

One can also set the advanced options for the *semantics* and *distributed computation* as document in *Advanced options*.

Results

The *Results* section displays a list of all previous verification batches.

Batch, run and query information

Clicking on a particular batch in the list of all batches provides detailed information and results regarding the selected batch. The detailed information includes

- a summary with general information such as the precise time, the verification time and the memory usage;
- all run options, regarding semantics, POR and distribution of computation;
- version information about the precise version of **deepsec** and the compiler.

These information are particularly useful for reproducibility, as well as for bug reporting.

Below the batch information it is possible to get information for each run. The information about a run contains the file name, number of queries and verification time.

Clicking on a run displays the result for each query. Additional information (semantics, type, ressources used) can be obtained by clicking on the query.

Details of a query: exploring attacks and equivalence proofs

By clicking on *Details* the tool provides additional information about the particular query. In particular it displays a summary of the query and recalls the signature (the function declaration and rewrite system), and in case of an attack, the attack trace.

It also displays the processes on which the verification was run and provides an equivalence, respectively attack simulator.

- The equivalence simulator allows the user to select a trace of one of the processes, by selecting a sequence of actions. The tool highlights available actions and the user chooses the next action. Once the user has selected a trace, one may request the tool to find an equivalent trace on the other process.
- The attack simulator allows the user to *replay* an attack: for an attack trace on one process, the user select actions on the other process that follow the attack trace. This allows the user to explore all possible available traces and convince the user that either no such trace exists, or that all traces lead to frames that can be distinguished.

Selecting the actions requires a few additional explanations:

- The user may select the level of details of the actions: `Default`, `I/O`, and `All`. When `I/O` is selected, only inputs and outputs are shown; internal τ actions are executed tacitly. When `All` is selected, the user also explicitly executes internal actions. `Default` is an intermediate choice where some internal actions are executed automatically, while others need to be selected explicitly. The `I/O` option is only available in case of an attack, not for an equivalence proof.
- Sometimes, when selecting an input, or an output, the user may choose between an internal communication (matching directly an input and output of the process on a same channel) and a communication with the attacker.
- When defining an attacker action in the equivalence simulator, it might be necessary to provide the attacker computation, called the recipe. This is required to provide the computation of the channel, for both inputs and outputs, as well as the term to be provided by the attacker for an input. These recipes are terms built from the (public) function symbols and constants of the signature, fresh names (prefixed by a `#`, e.g. `#n`) and elements of the frame, referred to as `ax_i`.

Settings

The settings allow to configure the **DeepSecUI** environment.

- *Show helpers* allows to turn on and off explanations that appear when you hover on options.
- *Absolute Path of deepsec_api* allows to set the path to the `deepsec_api` executable. This path is set automatically when the executable is in the PATH of your system environment.
- *Results directory* provides information where data on all runs and results are stored.
- The *Check API* button allows to test whether the `deepsec_api` executable is available at the path given above.
- The *Notification* section allows to configure the behaviour of pop-up windows. One can define the duration a pop-up window appears, which result notifications should be notified (batch, run, query), and define whether warning and error pop-ups should be "sticky", i.e., only disappear after manual removal. The behaviour can be tested using the *Test Notifications* button.

- The *Scan for new batch* button allows to scan for batches and runs that were run using the command line. These runs are then added to the list of batches in the *Results* section. When the `--title` option is used with the command line, the provided title will be used in the list.

Command-line options

Command-line options of DeepSec.

```
deepsec [OPTIONS] FILELIST
```

General options

`-s, --semantics VALUE (default=private)`

> Specify the default semantics of the process calculus. VALUE must be one of ‘private’, ‘classic’ or ‘eavesdrop’. See Section *Choosing the semantics* for detailed explanations.

`-p, --por BOOL (default=true)`

> Enable or disable Partial Order Reduction (POR) techniques for trace equivalence. BOOL must be either ‘true’ or ‘false’. Note that even when set to ‘true’, POR techniques only apply to action determinate processes. See Section *Partial order reductions* for detailed explanations.

`-t, --title TITLE`

> Set a TITLE for this run (displayed only with the graphical user interface).

`-q, --quiet`

> Only display the result of query verification, and no information about rounds.

`--trace`

> When an attack is found, display the full trace with the execution of the process. Incompatible with –quiet. **Not implemented yet**.

`-h, -help, --help`

> Display information about command line options.

Options for distributing computation

`-d, --distributed VALUE (default=auto)`

> Specify if the computation should be distributed. VALUE must be one of ‘auto’, ‘true’ or ‘false’. When VALUE=‘auto’, the number of workers will be set to the number of avilable physical cores. See Section *Distributing the computation* for detailed explanations. Note that when VALUE=‘true’, deepsec activates the distributed computation even if your computer only has one core.

`-l, --local_workers INT`

> Set the number of local workers to INT. If set, `--distriburted` is also set to ‘true’.

`-w, --distant_workers HOST PATH VALUE`

> Allows to add distant workers on machine HOST. See Section *Distributing the computation* for detailed explanations.
>
> - PATH must be the path on HOST to the directory that contains the deepsec executable.

> - VALUE must be either ‘auto’ or an integer, specifying the number of workers on HOST. When VALUE=‘auto’, the number of workers is set to the number of physical cores on the distant machine.
>
> Example: `-w login@my_server.server.org ~/deepsec/ auto`
>
> **Note**: It is possible to rely on multiple distant machine by using several instances of –distant_workers. Automatically sets –distributed to ‘true’.
>
> **Note**: the host must be configured with SSH key-based authentication.

`-j, --jobs INT`

> Specify the number of jobs to INT during the job creation phase. See Section *Distributing the computation* for detailed explanations. Automatically sets `--distributed` to ‘true’. The default number of jobs is 100 times the total number of workers.

`-r, --round_timer INT (default=120)`

> Sets the round timer to INT seconds. See Section *Distributing the computation* for detailed explanations.

Language reference

We describe in details the grammar of **DeepSec** input files. We use the following notations:

- X* denotes any (possibly zero) number of repetitions of X;
- {X} denotes that X is optional, i.e., zero or one occurence of X;
- seq X denotes a (possibly empty) comma separated sequence of X, i.e. seq X = {X (, X)*};
- seq+ X denotes a non-empty comma separated sequence of X, i.e. seq+ X = X (, X)*.

Moreover, we define the following types.

- <ident> is the set of identifiers that range over a sequence of letters (a-z, A-Z), digits (0-9), underscores (_), single-quotes (') where the first character of the identifier is a letter and the identifier is distinct from the reserved words of the language.
- <sem> is one of `classic`, `private`, `eavesdrop`.
- <int> is a natural number.

A file is a sequence of declarations (<decl>), process definitions (<proc_def>) and queries (<query>).

Terms and patterns

```
<term> ::= <ident>
         | (seq+ <term>)
         | <ident>(seq+ <term>)
```

```
<pattern> ::= =<term>
            | <ident>
            | (seq+ <pattern>)
```

Declarations

```
<decl> ::= set semantics = <sem>.
         | fun <ident>/<int> {[private]}.
         | const <ident> {[private]}.
         | free <ident> {[private]}.
         | reduc <term> = <term> (; <term> = <term>)*.
         | reduc <term> -> <term> (; <term> -> <term>)*.
```

Processes

```
<proc_def> ::= let <ident>(seq <ident>) = < process >.
```

```
<process> ::= 0
            | < ident>(seq <term>)
            | <process> | <process>
            | !^<int> < process>
            | (<process>)
            | in(<term>,ident); <process>
            | out(<term>,<term>); <process>
            | new <ident>; <process>
```

```
              | let <pattern> = <term> in <process> {else <process>}
              | if <term> = <term> then <process> {else <process>}
```

Queries

```
<query> ::= query trace_equiv (<process>,<process>).
          | query session_equiv (<process>,<process>).
          | query session_incl (<process>,<process>).
```

Comments

We allow 3 types of comments:

- all text following `//` on a given line is commented;
- all text between `/*` and `*/` is commented;
- all text between `(*` and `*)` is commented.

References

[ABF18] Martín Abadi, Bruno Blanchet, and Cédric Fournet. “The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication”. In: *J. ACM* 65.1 (2018), 1:1–1:41. DOI: 10.1145/3127586. URL: https://doi.org/10.1145/3127586.

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

[BCK20] Kushal Babel, Vincent Cheval, and Steve Kremer. “On the semantics of communications when verifying equivalence properties”. In: *Journal of Computer Security* 28.1 (2020), pp. 71–127. DOI: 10.3233/JCS-191366. URL: https://hal.inria.fr/hal-02446910/document.

[BDH15] David Baelde, Stéphanie Delaune, and Lucca Hirschi. “Partial Order Reduction for Security Protocols”. In: *Proc. 26th International Conference on Concurrency Theory (CONCUR’15).* Vol. 42. Leibniz International Proceedings in Informatics. Leibniz-Zentrum für Informatik, Sept. 2015, pp. 497–510. DOI: 10.4230/LIPIcs.CONCUR.2015.497.

[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.