0.a User guide

Building the program

It is required to have NodeJS 18 installed. Download the git repo: https://github.com/Sir-Sorensen/Seculizer

From the folder in the terminal run: `npm run production`

This will build everything from scratch and run it locally on the URL.

Figure 1: The front page of the application that will be opened on the 0.0.0.0:3000

How to create a protocol

A protocol contains eight sections: Participants, Knowledge, KeyRelations, Functions, Equations, Format, Icons, and Protocol where only Participants and Protocol are mandatory.

Participants

A protocol needs to have Participants and Protocol sections. In the Participants section, the Participants are defined and separated by a comma.

```
Participants: Alice, Bob;
```

A Participant can also have a comment by adding a question mark and a string or TeX representation after their identifier.

```
Participants: Alice ? "This is Alice", Bob ? $\text{This is Bob}$;
```

A Participant has a set of knowledge items, such as *msgA* or *key*, which they can modify and exchange with each other. This will be explained further later in the section about knowledge.

Protocol

In the Protocol section are the statements defined in the order they should be executed.

A statement can be sending a message, participant statements (such as creating new knowledge or setting the value of knowledge), matching input to do different statements and clearing knowledge.

Message send statements

A message can be sent in the format `sender -> receiver: msg;` The message can contain either raw content with a string, number, identifier or a function.

```
Participants: Alice, Bob;
Protocol:{
  Alice -> Bob: "Hello";
  Alice -> Bob: 100;
  Alice -> Bob: msgA;
  Alice -> Bob: hash(msgA, "stringParam");
}
```

The messages can also be encrypted by wrapping the message with curly brackets and giving a key afterwards. If the curly brackets are replaced with curly brackets and vertical bars then the message is signed with the given key.

```
1 Participants: Alice, Bob;
2 Protocol: {
3   Alice -> Bob: {"Encrypted message"}key; //This is a message that is
      encrypted with key
4   Alice -> Bob: {| "Signed message" |}key; //This is a message that is
      signed with key
5   Alice -> Bob: {"Multi", "Message", 50}key; //This message encrypts
      three messages
6   Alice -> Bob: {| "Multi", "Message", 50 |}key; //This message signs
      three messages
7 };
```

If Alice sends a message to Bob which is encrypted with a knowledge item then Bob also needs to know the knowledge item to decrypt the message.

A message can be both encrypted and signed. Furthermore, a message can be either signed or encrypted multiple times.

```
1 Participants: Alice, Bob;
2 Protocol: {
3   Alice -> Bob: {|{"Encrypted & Signed message"}key|}certificate; //
      This is a message that is encrypted with a key and signed with a
      certificate.
4   Bob -> Alice: {{"Double Encrypted message"}key1}key2; //This is a
      message that is encrypted with a key and with key2.
5 };
```

Participant statements

There are also types of participant statements. Participant statements have in common that they start with the identifier of the participant that should be modified followed by a colon. If new knowledge is needed then use the new keyword followed by the identifier or function that should be created. Strings and numbers are not supported since they are considered "Simple knowledge" that everyone has access to.

The value of a knowledge item can be set using the set statement which is the identifier followed by $:=$ and the new value.

```
1 Participants: Alice;
```

```
Protocol: {
  Alice: new nonce; //Creates a new nonce in Alice's knowledge
  Alice: nonce := 52; //Sets the value of the nonce knowledge instance
   in Alice's knowledge to 52
};
```

The new statement also supports giving the newly created knowledge a comment by following it with a question mark and either a string or TeX representation.

```
Participants: Alice;
Protocol: {
  Alice: new nonce ? "This is a nonce"; //Creates a new nonce in Alice'
   s knowledge with a displayed comment
};
```

Match statements

A protocol can also branch into different branches depending on the choice of the user. These choices are given with a match statement and are created by a participant sending a match statement to another participant

```
Participants: Alice, Bob;
Protocol: {
  Alice -> Bob: match {
    "Hallo Bob":{
      Bob -> Alice: "Hallo Alice";
    };
    "Goodbye Bob":{
      Bob -> Alice: "Goodbye Alice";
    };
  }; ? "Simple match case"
};
```

In this case, Alice can either send a hello message or a goodbye message to Bob.

In the UI the user will have two buttons to press with the content of “Hello Bob” or “Goodbye Bob” in them. If the user presses “Hello Bob” then Bob replies “Hallo Alice”.

Match cases also have the possibility to be given statement comments as seen above.

Clear statements

If the participants' knowledge gets too cluttered then a clear statement can be used to clear a piece of knowledge from all participants. This is commonly used for sessions where knowledge needs to be generated each time a session starts.

```
Participants: Alice, Bob;
Protocol: {
  Alice: new sessionId;
  Alice -> Bob: sessionId;
  clear: sessionId; //Clears sessionId from both Alice and Bob
};
```

Multiple pieces of knowledge can be cleared in the same statement by separating the identifiers with commas.

Knowledge Initial participant knowledge can be defined by using the Knowledge section. This section is always after the Participants section. The Knowledge section contains a list of all participants with their known knowledge. The knowledge items also support giving a comment by following it with a question mark and either a string or latex literal. These comments will show up on the knowledge item when a student hovers over them during visualisation.

```
Participants: Alice, Bob;
Knowledge: {
  Alice: aN, k ? "This is a key only Alice knows";
  Bob: bN;
};
Protocol: {
  Alice: aN := 52;
  Alice -> Bob: {aN}k;
};
```

Knowledge can also be shared by all participants by adding a "Shared" participant to the knowledge list:

```
Participants: Alice, Bob;
Knowledge: {
  Alice: aN;
  Bob: bN;
  Shared: k ? "This is a key all participants know";
```

```
};
Protocol: {
  Alice: aN := 52;
  Alice -> Bob: {aN}k;
};
```

KeyRelations

By default are all encryptions treated as symmetric encryption. An asymmetric key can, however, be defined by using the KeyRelations section. It takes a list, separated by commas, of tuples of secret (sk) and public keys (pk).

```
Participants: Alice, Bob;
Knowledge: {
  Alice: aSK;
  Bob: n;
  Shared: aPK;
};
KeyRelations: (sk:aSK, pk:aPK); //(pk:aPK, sk:aSK) would result in the
    same outcome
Protocol: {
  Bob -> Alice: {n}aPK;
};
```

When a public key is used during encryption then the secret key is needed to be known to be able to decrypt. In this case, Bob encrypts with Alice's public key and Alice can decrypt with the secret key in her knowledge.

Transparent and non-transparent functions

Calls to functions can always be called without being defined. However, the way a function is handled depends on the way it has been defined. A function can either be *transparent* or *non-transparent*. If a function is *transparent* then either the result of the function or the parameters of the function needs to be known, while if it is *non-transparent* then the result of the function needs to be known.

An example of this is the difference between an `add(x,y)` function and a `hash(x,y)` function. The `add(x,y)` function should be *transparent* since it is used as a utility to add knowledge items together. Thereby anyone should be able to compute the result if they know what x and y are. The `hash(x,y)` function on the other hand is

a *non-transparent* function and is used to compute a secret value and it should be unknown how to compute the value. This means that they only know the value if they know `hash(x,y)`.

By default are all functions *transparent* unless they are defined in the functions section. Here a list of function identifiers is given. Notice the comments for further explanations.

```
1 Participants: Alice, Bob;
2 Knowledge: {
3   Shared: x,y;
4 };
5 Functions: hash;
6 Protocol: {
7   Alice -> Bob: {"encrypted message"}hash(x,y); ? "This will not be
      decrypted by Bob since they don't know hash(x,y)"
8   Alice -> Bob: {"encrypted message"}add(x,y); ? "This will be
      decrypted by Bob since they know x and y"
9 };
```

Equations

The diffie-hellman protocol uses exponents to calculate shared keys. The exponents can be represented by `exponent(x,y)` which would be a representation of x^y. Since the exponent function is not defined in the functions section then it is a transparent function and the result can be calculated if a participant knows x and y. The problem in the Diffie-Hellman Key Exchange protocol is that it uses the exponent function recursively to say $(g^x)^y$ or $(g^y)^x$ depending on which participant encrypts. However, if Alice sends `exponent(exponent(g,y),x)` to Bob where g is the generator, x is Alice's private key and y is Bob's private key. Bob only knows exponent(g,x) and y, which means they cannot create the exponent(exponent(g,y),x) key since they don't know what x is. However, by the definition of the exponent rule then x and y can be flipped. An equation can be used to make sure that $(g^X)^Y$ is the same as $(g^Y)^X$. Therefore an equation is needed to be defined for exponents saying that for any function in the format `exponent(exponent(a,b),c)` then it can also be equal to `exponent(exponent(a,c),b)`.

```
1 Participants: Alice, Bob;
2 Knowledge: {
```

```
  Alice: exponent(x,z),y;
  Bob: exponent(x,y), z;
  Shared: x;
};
Equations: exponent(exponent(a,b),c) => exponent(exponent(a,c),b);
Protocol: {
  Alice -> Bob: {"encrypted message"}exponent(exponent(x,z),y); ? "This
    will be decrypted by Bob since he knows exponent(x,y) and z"
};
```

Format

To improve visual representations of types then a format can be given. This is for example the case if you want to display `exponent(a,b)` as a^b using TeX, or display Alice's public key, *apk*, as pk_{Alice}, or *sid* as "Session ID". A format takes an identifier or function and formats that to a string or TeX representation.

```
Participants: Alice, Bob;
Knowledge: {
  Alice: apk,sid;
  Bob: x,y;
};
Format: apk = $pk_{Alice}$, sid = "Session ID", exponent(a,b) = $a^b$;
Protocol: {
  Alice -> Bob: {|sid|}apk;
  Bob -> Alice: exponent(x,y);
};
```

Icons

All identifiers and function identifiers can be assigned an icon from the Openmoji library (https://openmoji.org/library/) or an emoji. If an emoji is given then an Openmoji representation will be used to standardise the icons. Openmoji icons are given using the name of the emoji with spaces replaced with dashes. For example, the grinning face icon should be given as *grinning-face*. The icons are given by having a set of icons and assigning identifiers to the icons such as in the example below:

```
Participants: Alice, Bob;
Knowledge: {
```

```
  Alice: apk,sid;
  Bob: x,y;
};
Format: apk = $pk_{Alice}$, sid = "Session ID", exponent(a,b) = $a^b$;
Icons: {
  "woman": Alice;
  "man": Bob;
  "key": apk;
  "input-numbers": x, y, exponent;
};
Protocol: {
  Alice -> Bob: {|sid|}apk;
  Bob -> Alice: exponent(x,y);
};
```

In this case, Alice and Bob are given the "woman" (👩) and "man" (👨) icons respectively and the "key" emoji (🔑) is given to *apk* which is the only key in the protocol. X and Y are assumed to be numbered and will thereby be given the "input-numbers" icon (🔢). The same goes for all functions that have the identifier *exponent*.

Example of Diffie-Hellman Key Exchange protocol:

```
Participants: Alice, Bob;
Knowledge: {
    Alice: X ? "This is a nonce", msgA ? "This is Alice's message";
    Bob: Y ? "This is a nonce", msgB ? "This is Bob's message";
    Shared: g ? "This is a generator";
};
Equations: exp(exp(a,b),c) => exp(exp(a,c),b);
Format: exp(x,y) = $x^y$;
Icons: {
    "woman": Alice;
    "man": Bob;
    "input-numbers": X, Y;
    "key": exp;
    "gear": g;
    "memo": msgA, msgB;
};
Protocol: {
    Alice -> Bob: exp(g,X); ? "Alice sends Bob her public key"
```

```
19      Bob -> Alice: exp(g,Y); ? "Bob sends Alice his public key"
20      Alice -> Bob: {msgA}exp(exp(g,X),Y); ? "Alice sends Bob a message
        encrypted with Bob's public key"
21      Bob -> Alice: {msgB}exp(exp(g,Y),X); ? "Bob sends Alice a message
        encrypted with Alice's public key"
22  };
```

Listing 1: Diffie-Hellman Key Exchange in the sepo-lang syntax

How to view a protocol

When you have written or uploaded a complete security protocol, click the "Generate" button to generate the interactive slideshow. This will create a generated link, which can be shared with others with the program. Anyone can view your protocol by opening the generated link. If the URL is *localhost:3000* and a generated protocol with the value `AoQwTgLglgxlAOIB2EDOAuABCA3AKGDAHsIiYiAbLAbz02yyQFMB3TVfAXxyA` then you can open it on *localhost:3000/prototype/AoQwTgLglgxlAOIB2EDOAuABCA3AKGDAHsIiYiAbLAbz02yyQFMB3TVfAXxyA*