Bachelor Thesis

A user interface for interactive security protocol design

Author(s): Staub, Cedric

Publication Date: 2011

Permanent Link: https://doi.org/10.3929/ethz-a-007554662

Rights / License: In Copyright - Non-Commercial Use Permitted
Bachelor thesis

A user interface for interactive security protocol design

Cedric Staub
cstaub@ethz.ch

Supervisor: Simon Meier
Professor: Prof. D. A. Basin
Issue Date: February, 21st 2011
Submission Date: August, 10th 2011
# Contents

1 Introduction .............................................. 5

2 Background .............................................. 6
   2.1 Prover .............................................. 6
   2.2 Technologies ....................................... 6

3 User interface ........................................... 7
   3.1 Overview ........................................... 7
   3.2 Working with proofs ................................ 7

4 Implementation ......................................... 10
   4.1 Overview ........................................... 10
   4.2 Server modules .................................... 10
   4.3 Client/Server interaction ......................... 20

5 Case study ............................................... 23
   5.1 Overview ........................................... 23
   5.2 The key-exchange protocol ......................... 23
      5.2.1 Mathematical description ..................... 23
      5.2.2 Theory file for the prover .................... 23
   5.3 Defending against an external adversary ........... 26
   5.4 Defending against an active adversary ............. 26
   5.5 Adding session-key compromise capabilities ........ 29
   5.6 Lessons learned ................................... 30

6 Conclusion .............................................. 31

7 Bibliography ............................................ 33

A Additional figures ...................................... 35
1 Introduction

This thesis describes the design and implementation of a graphical user interface for security protocol design. It can be read both as a user guide for the resulting interface as well as a full thesis. If you want to read this thesis as a user guide, only read the introduction and the section on the user interface.

**Problem context** Traditionally, cryptographic security protocols have been designed by specifying their message exchange, informally stating what properties this message exchange should achieve, and giving an informal argument why these properties are always satisfied. However, it turned out that these informal arguments were often wrong and the protocol was broken. Therefore, methods and tool support have been developed to provide formal proofs for the expected security properties.

Usually these tools take a formal definition of a security protocol and its desired properties as input. They then output a concrete counter-example if a security property is not satisfied. Conversely, if a security property is satisfied, they output a message noting that the property is satisfied for this protocol. This does not help the designer of the protocol understand why a security property is satisfied however. Such an understanding is required to make informed choices about changes to the protocol (e.g., minimizing the size of the messages to make the most efficient use of available bandwidth) or the environment (e.g., the impact of composing the protocol under investigation with additional protocols). We will be referring to such tools as provers throughout the rest of the thesis.

In recent work [6], Meier, Cremers, and Basin extended the algorithm underlying the Scyther prover to produce short, human-readable proofs that explain formally why a protocol satisfies a security property. The corresponding implementation is written in Haskell and searches automatically for proofs. Once a proof is found, its proof steps and the corresponding proof states are visualized as an HTML page. Hence, the protocol designer can inspect and understand the proofs and infer the impact of changes to the protocol on the validity of the security properties and their current proofs.

When using the inference system proposed in [6] to construct proofs, such an approach can even be incremental, as, in most cases, small changes to the protocol only invalidate small parts of the proofs of its security properties. Until now, the tool only offered a batch mode, which took a specification of the security protocol and its desired properties as input and gave the results as output. This means that after every change in the protocol, the tool had to be re-run and the output examined separately once again. The goal of this thesis was to develop an interactive user interface in order to lift this restriction.

**Approach** The approach taken was to develop a stand-alone web application integrated into the implementation of the ideas described in [6] developed by Meier and Schmidt. Integration with this implementation allows for direct reuse of the existing proof infrastructure and visualization code. The client and server were developed in tandem and are directly shipped with the implementation. The server is embedded in the prover implementation and as such also written in Haskell. The user interface is in part also embedded in the implementation, and in another part shipped as separate files alongside the rest of the code.

**Contributions** The first contribution of this thesis is the user interface for the prover. The interface allows for direct interaction with the prover. Instead of having just a static visualization of the output of the prover, the user can now interact with and modify what he sees. Proofs can be manually constructed by the user or automatically generated by the prover. Individual parts of the proof can be modified, removed and reconstructed. Visualizations of various parts of the proof can be explored by clicking on the corresponding proof steps. New security properties can be added and proven on-line. The interface also allows for editing the protocol iteratively and adapt the proofs. All steps can be reverted using the built-in undo/redo functionality through the browser's back/forth buttons.

The second contribution is a case study where a simple security protocol is designed incrementally using the user interface. The goal is to test the resulting implementation and how well the resulting
workflow matches the initial expectations. It also shows problems and ideas on how the idea of incremental security protocol design could be expanded on in the future.

2 Background

2.1 Prover

The implementation of the user interface integrates with the implementation by Meier and Schmidt mentioned in section 1. Their implementation is called dh-proto-proof and is written in Haskell.

The prover models a security protocol interacting with a Dolev-Yao style intruder. The Dolev-Yao model is a formal model for proving properties of security protocols. The attacker in the model is allowed to overhear messages, intercept them, and forge new ones. The only limitation are cryptographic constraints - cryptography is assumed to be perfect. For example, the attacker can not recover the plaintext of an encrypted message without the corresponding key.

The state of the protocol and the intruder is modeled as a multiset. A multiset is a generalized set in which members can appear more than once. The state multiset contains so-called facts. A fact is, in simplified terms, either a ground message or a message derived from a ground message. A ground message could for example be a fixed string, with the encryption of that string being a derived message. A fact can also be protocol specific. For example, a protocol model could introduce an new fact for keeping track of session keys.

The protocol itself is modeled as a set of rewriting rules on the multiset that models the state of the protocol. Such a rule can either be applicable to the empty set or have prerequisite facts. Rules with no prerequisites can always be applied, rules with prerequisites can only be applied if all the required facts are in the multiset. Application of a rule can rewrites its prerequisite or introduce new facts into the multiset. A set of multiple such rules is called a multiset rewriting system.

As an example, consider a protocol with two steps for a given role $R$. The first step can be modeled as a rule which has no prerequisites. This rule can always be applied by the prover. In other words, a new thread executing the role $R$ can always be started, regardless of the current state. To note that the first step of the protocol was executed, the rule can introduce a custom fact into the state. The second step of the protocol can then be modeled as a rule which requires this custom fact introduced by the first rule as a prerequisite fact. This ensures that the second step can only be executed after the first one. It also allows to keep track of state for the execution of each role.

A few special rules are built into the model by default. One example would be the Fresh rule, which can be used to introduce new so-called fresh values. A fresh value models a value generated by a random number generator and is guaranteed to be unique within the model. In other words, the random number generator is assumed to be perfect. The probability that it generates repeated values is considered to be negligible. Another such rule would be Knows, which is used to denote values which are known to the intruder. For example, the fact Knows($k$) denotes that the intruder has knowledge of the value $k$.

Verification of the protocol is then performed within this model. Security properties which should be verified must be expressed in terms of the model. For example, session-key secrecy could be verified by expressing that if a session key $k$ has been derived by the protocol the fact Knows($k$) may never enter the multiset. A specific example of this can be found in the case study in section 5.

2.2 Technologies

The user interface was developed as an interactive web application. A web application consists of two major parts, a client and a server.

The client runs inside a browser and is written in multiple languages. The first one would be Hypertext Markup Language (HTML), which describes the structure of the graphical interface.
With Cascading Style Sheets (CSS), a layout can be applied to the structure described by HTML. For example, CSS can be used to set fonts, font sizes, colors, ordering, spacing, borders and many more properties that describe how the HTML code should be rendered. And finally there is JavaScript, a scripting language which runs inside the browser and can be used to dynamically update the interface, query new information from the server, respond to user-generated events, and others. JavaScript is used to make the interface interactive, as opposed to a static rendering of HTML code.

The second major component of any web application is the server. The server communicates with the client using Hypertext Transfer Protocol (HTTP). The server component is responsible for serving data to the client and exposing all functions which are needed by the client. The server also bootstraps the client by serving all client code to the browser.

There are currently two major web frameworks available in Haskell, Snap [2] and Yesod [3]. Upon comparing both frameworks, none of them clearly came out as a winner. Both frameworks have similar features, both are being actively developed, both have active communities built around them, and so forth. In the end, I decided to use Yesod because I was already somewhat familiar with it.

The Yesod web framework tries to apply Haskell’s strengths to the realm of web programming. This makes for a robust web framework: Yesod is secure and type-safe, preventing various attacks like cross-site scripting, cross-site request forgery and SQL injection directly on the framework-level [1]. These attacks are very common in modern web applications [4], so having a framework which directly prevents them is certainly welcome.

Yesod also contains various additional helper packages like Hamlet. Hamlet is a type-safe HTML template language. It can be used to write Haskell code that dynamically generates HTML code. Other helper packages include functions to deal with forms, static routing, and various other aspects of web development.

3 User interface

3.1 Overview

In order to use the server one can simply open up a command line terminal and type dh-proto-proof interactive working-directory. The working-directory is the path to the directory where one would like to keep all theory files, generated images, and other related data. The server will then start up and try to load any theory files that it can find in the working directory.

After the server is running, the interface can be accessed by opening a browser on the local machine and accessing http://localhost:3001. This is the address where the server is listening by default. The server will also print the exact address on the command line after startup.

The loaded theory files are displayed on the first page of the interface. One can load and start editing a theory by clicking on it and making modifications to it in the interactive interface. Theory files that were loaded at startup can then also be saved back to disk.

The basic workflow can be described as follows: First, write an initial formal specification of the protocol. Second, attempt verification of desired properties using the prover. Third, Analyze results. If necessary make modifications and go back to the second step.

3.2 Working with proofs

From a graphical perspective the interface is divided into multiple sections, as can be seen in figure 1. By default we have the proof script on the left side, and the main view on the right side. On top we have the toolbar.

In the proof script on the left side we can see the name of the theory (in this case CR Paper), links to the rewriting rules and source case distinctions, as well as a list of lemmas that were defined.
In this case there is only one lemma with the name \texttt{C_k_secrecy}. Below the lemma we can see the proof for the lemma. At the moment, the lemma is unproven.

As can be seen by the yellow highlighting, the first step of the proof that tells us that the lemma is unproven is selected. Proof steps will change their background color based on their status. The currently selected step has a yellow background. Proven steps have a green background. Unproven, unselected steps have no background. Invalid steps have a red background. Invalid steps can arise for example after editing a theory, which can render parts of a proof invalid because the rules used in the steps have changed.

On the main view in the right side we can see the theory path that is currently selected. Figure 1 shows us the graph part of the sequent, and below we get a list of applicable proof methods that we can select in order to construct a proof.

The interface can do more than figure 1 shows. For example, if we don't want to construct a proof manually, we can call the autoprover and let him try to construct a proof for us. This can be done by right-clicking on a proof step to reveal a context menu seen in figure 2. Through the context menu it is also possible to remove a proof step, for example to manually edit and optimize parts of a proof constructed by the autoprover.
Figure 3: The actions and options menus in the toolbar.

Figure 4: The full interface including an open debug pane.

We can also perform various actions using the toolbar. Besides saving and downloading a theory, it is also possible to directly look at the source, edit the theory, or add a new lemma interactively. The open menus can be seen in figure 3.

Under the options menu we can also change the way graphs are rendered, or pull up a debug pane on the right in order to display more information about the current proof step in order to help debugging the prover. The green/red bullets on the right side indicate whether an option is currently enabled or disabled. In this case, compact graphs and compressed sequents are enabled, but the debugging pane is hidden. Enabling/disabling an option automatically reloads the main view with the new options set.

After having applied the autoprover on the secrecy lemma and having pulled up the debug pane for example, the interface looks like displayed in figure 4. Note that the proof steps are highlighted with a green background to signify that this proof is complete. Also note that we are displaying the last step of the proof in the main view, where one can see the corresponding graph. In the debug pane on the right, internal information is shown that can be used in order to debug the server or prover code.

Another notable feature of the user interface are the keyboard shortcuts. For example, the \( j \) and \( k \) keys can be used in order to navigate up/down between proof steps, and the 0-9 keys can be used in order to apply proof methods to the currently selected step. In addition, the keys \( J \) and \( K \) have been implemented as smarter versions of \( j \) and \( k \). They jump to the next/previous unproven
step in the proof respectively. In other words, they can be used to quickly navigate to the parts of the proof that require the users attention.

4 Implementation

4.1 Overview

The tool was implemented as a web application for various reasons. Compatibility with multiple platforms is one major factor. All that is needed in order to run the user interface is a browser. Multiple competing browser implementations exist for all major operating system platforms. This makes it possible for the user interface to run on many different operating systems without having to make modifications in the underlying codebase. It also decouples the frontend and the backend. This allows for running the server on a different machine than the client, for example over the internet.

The server is directly embedded in the prover. This makes it very easy for the end-user because everything can be controlled with just one executable. Since the prover is written in Haskell the server must also be written in Haskell to make this possible. Haskell is a purely functional, statically-typed programming language. As Haskell is a functional programming language programs written in Haskell do not consist of a sequence of commands. A program in Haskell is an expression. It is executed by evaluating that expression.

Haskell programs are structured using a hierarchical modules.

Some familiarity with Haskell is assumed. If that is not the case, the free online book Real World Haskell [8] provides a great resource for learning Haskell.

4.2 Server modules

The server is split into multiple Haskell modules with different purposes. In this section, we will look at all the modules and explain what they do. The server consists of the following Haskell modules.

1. Web.Types: Data types, type synonyms, typeclass instances.

Note that interaction with dh-proto-proof is done through the Web.Theory module, whereas the other modules are mostly independent of the rest of the dh-proto-proof infrastructure. This can be seen in figure 5, which shows the dependency graph of the modules in dh-proto-proof. Note that most parts of the graph have been collapsed for brevity, only the Web modules are shown in full detail.

Web.Types

The Types module is the central module of the server. It contains all the important data types, type synonyms, typeclass instances, and routing information for the server. I will now list various important sections of code in this module, starting with some type synonyms.
The first four types synonyms were defined simply in for convenience. Yesod defines the notions of handlers and widgets. Handlers are functions that handle a request made to the server. Widgets are simply a pieces of data that can be displayed in a browser. In other words, a widget contains all the necessary information (HTML/CSS/JavaScript) necessary for rendering something to the user.

Both handlers and widgets are associated with a so-called site. A site can be thought of as a data type which can contain stateful information about the server. For example, the site data type could contain functions which allows handlers to access a database. In our case, the site datatype contains functions that allow manipulating theories.

Our server only uses handlers running on top of our site datatype WebUI. Since we don't want to spell that out everytime we write a handler, we use the type synonyms Handler and GenericHandler. Both are monadic types. A generic handler can run in an arbitrary monad. A handler always runs on top of an enumerator on top of IO.
The WebUI is the site data type, or site argument, for our server. It holds various information that is needed by the handler functions at runtime. It contains settings for static file serving, the path to the working directory, and functions for storing/retrieving theories.

The site data type also contains various functions for dealing with threads: In order to prevent lockups, handlers can evaluate values in separate threads which can then be terminated by the client without terminating the main process. The implementation of this mechanism can be found in the functions evalInThread and future in the WebDispatch module.

| Simple data type for generating JSON responses.

```haskell
data JsonResponse =
  JsonAlert T.Text
  | JsonRedirect T.Text
  | JsonHtml T.Text
```

The JsonResponse data type that is mainly used for server/client interaction. On the server side this data type is used in order to send back a response to an asynchronous request the client made. The response is then parsed by the client, which takes the appropriate action. The server response can either be new HTML code for the main view, an message to the user (called an alert), or the server can tell the client to redirect to a new page.

| Data type containing both the theory and it's index, making it easier to pass the two around (since they are always tied to each other). We also keep some extra bookkeeping information.

```haskell
data TheoryInfo
  = TheoryInfo
    { tlIndex :: TheoryIdx
    , tlTheory :: ClosedTheory
    , tlTime :: ZonedTime
    , tlParent :: Maybe TheoryIdx
    , tlPrimary :: Bool
    , tlOrigin :: TheoryOrigin
    }
```

The TheoryInfo data type that can hold information about a theory that the server is handling. It contains everything the handler functions need in order to deal with theories. Next, let's take a look at how we map requests to handlers.
The `mkYesodData` instruction is a piece of Template Haskell code which creates an instance of the typeclass `YesodData` for our WebUI automatically based on the routing information provided. Let us take a look at one of the lines in more detail in order to explain what is happening here.

```haskell
/thy/#1 substantive/route TheorySourceR GET
/thy/#1 substantive/variants TheoryVariantsR GET
/thy/#1 substantive/main/MP(TheoryPath) TheoryPathMR GET
/thy/#1 substantive/debug/MP(TheoryPath) TheoryPathDR GET
/thy/#1 substantive/graph/MP(TheoryPath) TheoryGraphR GET
/thy/#1 substantive/autoprove/MP(TheoryPath) AutoProveR GET
/thy/#1 substantive/prev#/String/MP(TheoryPath) PrevTheoryPathR GET
/thy/#1 substantive/save SaveTheoryR GET
/thy/#1 substantive/download/#String/DownloadTheoryR GET
/thy/#1 substantive/edit/source EditTheoryR GET POST
/thy/#1 substantive/edit/path/MP(TheoryPath) EditPathR GET POST
/thy/#1 substantive/delete/path/MP(TheoryPath) DeleteStepR GET
/thy/#1 substantive/unload UnloadTheoryR GET
/thy/#1 substantive/kill KillThreadR GET
/thy/#1 substantive/threads ThreadsR GET
/thy/#1 substantive/robots.txt RobotsR GET
/thy/#1 substantive/favicon.ico FaviconR GET
/thy/#1 substantive/static StaticR Static getStatic
```

Let us take a look at one of the lines in more detail in order to explain what is happening here.

```haskell
The co de above describ es the p ossible theo ry paths. F o r example, a theo ry path could p oint around the
```

The code describes the possible theory paths. For example, a theory path could point to a part of a proof of a lemma. This is the `TheoryProof` case. Two pieces of information are necessary to resolve such a path. First we need the name of the lemma, which is the `String` component in the constructor. Then we need the path within the proof, which is the `ProofPath` component. In this case, a `ProofPath` is a list of `Strings`. Each `String` in this list describes which step to follow.

The `Web.Types` module also provides the functions `renderPath` and `parsePath` which are used to render/parse a theory path to/from a list of strings. The function `joinPath` is used to join a list of strings to a single string representing a path with a single slash as a delimiter. The `joinPath` function also performs the necessary escaping. A prime has been added to the name in order to avoid a naming conflict with Yesod, which has a `joinPath` function for joining paths within Yesod.

The `Web.Types` module also contains a couple of helper functions and other things necessary for dealing with the data types defined in the module (like typeclass
instances for example).

Web.Dispatch

The dispatch module is the module which is called from Main. It is responsible for bootstrapping and initializing the components the server needs in order to run. The functions for loading, storing, retrieving theories and managing threads can be found in this module.

<table>
<thead>
<tr>
<th>Initialization function for the web application.</th>
</tr>
</thead>
<tbody>
<tr>
<td>withWebUI :: FilePath</td>
</tr>
<tr>
<td>-&gt; (FilePath -&gt; IO ClosedTheory)</td>
</tr>
<tr>
<td>^ Working directory.</td>
</tr>
<tr>
<td>-&gt; (String -&gt; IO ClosedTheory)</td>
</tr>
<tr>
<td>^ Theory loader (from file).</td>
</tr>
<tr>
<td>-&gt; (OpenTheory -&gt; ClosedTheory)</td>
</tr>
<tr>
<td>^ Theory loader (from string).</td>
</tr>
<tr>
<td>-&gt; Bool</td>
</tr>
<tr>
<td>^ Theory closer.</td>
</tr>
<tr>
<td>-&gt; Maybe FilePath</td>
</tr>
<tr>
<td>^ Show debugging messages?</td>
</tr>
<tr>
<td>-&gt; (Application -&gt; IO b)</td>
</tr>
<tr>
<td>^ Path to static content directory</td>
</tr>
<tr>
<td>-&gt; IO b</td>
</tr>
<tr>
<td>^ Function to execute</td>
</tr>
</tbody>
</table>

The withWebUI function is the only function that is exported from this module. It receives all the information needed for running the server through arguments, instantiates a WebUI object, and calls a given function with the object as an argument.

As can be seen on the first two lines of the function, we first create two new MVars which we use in order to store the maps holding the theories the server is handling as well as information about threads that are currently running.

The WebUI is then only handed functions that allow manipulation of the contents of the map within the MVar, but never a reference to the MVar itself. This hides the actual details of how the state is stored from the handler functions.

For example, it would be possible to replace the implementation of the theory storage component with a new implementation which stores theory files directly on disk instead of in memory. The handler functions do not, and do not need to know, how theories are stored, making it possible to replace the current implementation with something more suited should the need to do so arise.

Web.Handler

The handler module contains all custom handler functions, which are used in order to process requests like displaying, storing, modifying theories. This module also provides functions for evaluating values in separate threads. The handlers correspond to the routes defined in the previous module. In order to pick up from our previous example (the route definition for retrieving the main view), here is the corresponding handler function which handles GET requests for the TheoryPathMR route:

<table>
<thead>
<tr>
<th>Show a given path within a theory (main view).</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTheoryPathMR :: TheoryIdx</td>
</tr>
</tbody>
</table>
The type of the function corresponds to the route that was defined: the first argument being an index to a theory (an integer), and the second argument being a path in the theory. As can also be seen from the type, the function is a handler (it runs within the 

```
GetTheoryPathMR idx path = liftIOHandler $ do
  jsonValue <- evalInThread $ withTheory idx (go path)
  return $ RepJson $ toContent jsonValue
```

where

- Handle method paths by trying to solve the given goal/method
- go (TheoryMethod lemma proofPath i) ti = modifyTheory ti
  (
    \(\text{\textbackslash{}thy} \rightarrow \text{applyMethodAtPath \\text{\textbackslash{}thy} lemma proofPath i}\)
    JsonAlert "Sorry, but the prover failed on the selected method!"
  )
- Handle generic paths by trying to render them
- go _ _ ti = do
  let title = T.pack $ titleTheoryPath (ti Theory ti) path
  let html = T.pack $ renderHtmlDoc $ htmlTheoryPath (ti Theory ti) path
  return $ responseToJson (JsonHtml title (toContent html))

As can be seen, the type of the function corresponds to the route that was defined: the first argument being an index to a theory (an integer), and the second argument being a path in the theory. As can also be seen from the type, the function is a handler (it runs within the Handler monad) and it returns a response in the JavaScript Object Notation (JSON).

The function also uses the custom `evalInThread` function, which wraps a computation and evaluates it in a separate thread. This is a central function in this module and is used in most other handlers as well.

```
| Fully evaluate a handler returning a JSON response.
| evalInThread :: (NFData a, MonadControlIO m) => GenericHandler m a
|   GenericHandler m a
|   evalInThread handler = do
|     yesod <- getYesod
|     renderF <- getUrlRender
|     maybeRoute <- getCurrentRoute
|     case maybeRoute of
|       --- Normal response, run in thread
|       Just route -> do
|         let key = renderF route
|         (killFunc, readFunc) <- future handler
|         putThread yesod key killFunc
|         res <- readFunc 'E.finally' delThread yesod key
|         case res of
|           Nothing -> notFound
|           Just value -> return value
|           --- Error response, just run it normally
|           Nothing -> handler
```

As can be seen in the type, this function wraps a `GenericHandler` on top of a monad which implements `MonadControlIO` and returns a value which has an instance of `NFData` so that the handler is evaluated in a distinct thread. The function also registers the thread in the global map of threads such that it can be killed when the client requests it.

We need the constraint that the monad needs to implement `MonadControlIO` so that we can catch any exceptions that the handler might throw. We want to avoid specifying `IO` as the only allowed monad, because other monads are fine as long as we can catch exceptions.

We also need the `NFData` constraint so that we can guarantee that the value is fully evaluated within the scope of the thread. Otherwise, due to lazy evaluation, the thread could get away by returning a thunk and offloading the evaluation to the original thread. This we want to avoid.

```
| Evaluate action in thread, handle exceptions appropriately.
| Returns action to terminate computation and action to read result.
```
The `future` function is the actual heart of the threading code: This function spawns another thread, runs the handler, fully evaluates the response, writes it to an `MVar`, and handles any exceptions that might arise along the way. We also make sure that a thread can only be killed once using the provided `kill` function, in order to avoid accidentally killing another thread after the thread id has been recycled.

The `liftControlIO` function from the `MonadControlIO` typeclass is used in order to implement a generalized version of `forkIO`.

And finally, we also need a handler function which can be used by the client side in order to make a request that a thread spawned by a previous request under a given path should be terminated:

```haskell
future :: (MonadControlIO m, NFData a) => m a -> m (IO (), m (Maybe a))
future action = do
  mv <- liftIO newEmptyMVar
  killMv <- liftIO newEmptyMVar
  let handler :: MonadControlIO m => E.SomeException -> m ()
      handler = liftIO . putMVar mv . Left
  tid <- liftForkIO $ E.handle handler $ do
        res <- action
        E.evaluate (mnf res)
        liftIO $ putMVar mv (Right res)
  return (killFunction killMv tid, readFunction mv)
where
  — Kill thread with given thread id, make sure we only kill it once
  — by filling an MVar to signal that the thread was killed.
  killFunction mv tid = do
    alive <- tryPutMVar mv ()
    when alive $ killThread tid
  — Try to read result of action from the corresponding MVar, and make
  — sure to re-throw exceptions (except ThreadKilled exceptions).
  readFunction mv = do
    res <- liftIO $ readMVar mv
    case res of
      Left e -> E.handle handleAsync $ E.throwIO e
      Right x -> return (Just x)
  — Catch ThreadKilled exceptions, re-throw others (e.g. stack overflow).
  handleAsync :: MonadControlIO m => E.AsyncException -> m (Maybe a)
  handleAsync E.ThreadKilled = return Nothing
  handleAsync other = E.throwIO other
  — Helper function: Generalized version of forkIO
  liftForkIO :: MonadControlIO m => m a -> m ThreadId
  liftForkIO m = liftControlIO $ \runInIO -> forkIO $ void $ runInIO m
```

The `kill` function (aka `cancel request`).

```haskell
getKillThreadR :: Handler RepPlain
getKillThreadR = liftIOHandler $ do
  maybeKey <- lookupGetParam "path"
  case maybeKey of
    Just key -> do
      trace ("Killing: " ++ T.unpack key) $ tryKillThread key
      return $ RepPlain $ toContent ("Canceled request!" :: T.Text)
    Nothing -> invalidArgs ["No path to kill specified!"]
where
  tryKillThread k = do
    yesod <- getYesod
    maybeKillFunc <- getThread yesod k
    E.finally
      { case maybeKillFunc of
        Nothing -> return ()
        Just killFunc -> trace
        ("Killing thread " ++ T.unpack k)
          (liftIO killFunc)
      (delThread yesod k)
```
The `getKillThreadR` receives a path as an argument and looks up the path in the map of currently registered threads. If found, it calls the kill function associated with the thread in order to terminate it.

Of course, the handler module is responsible for more than just handling threads. There are various handlers which need to run forms or modify loaded theories. Instead of showing all those handlers here, I decided to show the common high-level functions that they are all implemented with.

--- | Run a form and provide a JSON response.

```haskell
formHandler :: (HamletValue h, HamletUrl h "WebUIRoute", h "Widget ()")
  => T.Text
  ^ The form title
  => Form WebUIWebUI a
  ^ The formlet to run
  => (Widget () => Enctype => Html => h)
  ^ Template to render form with
  => (a => GenericHandler IO repJSON)
  ^ Function to call on success
```

`formHandler title formlet template success = do`

```haskell
  (result, widget, enctype, nonce) <- runFormPost formlet
  case result of
    FormMissing => do
      repHtmlContent <- ajaxLayout (template widget enctype nonce)
      jsonResponse $ jsonHtmlTitleContent
    FormFailure => jsonResponse $ showAlert
    "Missing fields in form. Please fill out all required fields."
    FormSuccess res => do
      liftIOHandler (success res)
```

The `formHandler` function is a helper function which can be used to run forms very easily. I implemented this after I noticed that all my form-handling functions follow the same pattern: First run the form, and then render the form if it wasn’t submitted, show an error message if it failed, or run some given function if it succeeded.

So this function has various arguments: The title to use for the form, the formlet (a data type representing a form) to run, a template to display the form, and a function to call if the form ran successfully. All other form handlers are implemented using this function and simply setting the appropriate parameters.

--- | Modify a theory, redirect if successful.

```haskell
modifyTheory :: (MonadIO m, Functor m)
  => TheoryInfo
  => (ClosedTheory => Maybe ClosedTheory)
  => JsonResponse
  => GenericHandler m Value
modifyTheory ti tf errResponse = do
  case f (tiTheory ti) of
    Nothing => return (responseToJson errResponse)
    Just thy => do
      yesod <- getYesod
      newThyIdx <- putTheory yesod (Just ti) Nothing thy
      newUrl <- getUrlRenderer <$> pure (OverviewR newThyIdx)
    return , responseToJson $ JsonRedirect newUrl
```

The `modifyTheory` function can be used in order to modify a theory using a given function. It takes information about the theory to be edited as the first argument, and then a function to call on the theory as well as a response in case something should go wrong as second and third arguments. All other functions which modify a theory are implemented using this one, as can be seen in the earlier example of the handler for the main view.

Last but not least we will look at how graphs are handled, and how we make it possible to request different forms of graphs (compact/uncompressed and compact/uncompact).

--- | Get rendered graph for theory and given path.

```haskell
getTheoryGraphR :: TheoryIdx => TheoryPath => Handler ()
getTheoryGraphR idx path = withTheory idx $ \ti => do
  yesod <- getYesod
```
The parameters "uncompact" and "uncompress" are set by the client side within the JavaScript component. By default, we serve images with compressed sequents and compact graph representation. Then we load the graph from the working directory that is associated with the requested path by calling `pngThyPath` from the `Web.Theory` module. Then we send back the file once we found it.

**Web.Theory**

The `Web.Theory` module is the module which interfaces with the prover module of `dh-proto-proof`. It is responsible for pretty-printing theories to HTML code, generating images for displaying the graphs, applying proof methods at a given path among others.

The `Web.Types` module makes use of two modules from the prover: The `Theory` module and the `Theory.Pretty` module. The `Theory` module exposes the basic interface required to deal with theories: The basic data types and type synonyms and functions to extract information from the theories and make basic modifications. The `Theory.Pretty` module exports all the pretty-printing code that we need in order to display theories. Let us take a look at an example function in this module.

```haskell
render
```

Render the item in the given theory given by the supplied path.

```haskell
htmlThyPath :: HtmlDocument d => TheoryPath thy path = go path

where

go TheoryRules = rulesSnippet thy

```

The `htmlThyPath` one of the most central functions in this module. It is used by the rest of the server in order to pretty-print a given path within a theory for the main view. As can be seen, the function first decodes the theory path and depending on what kind of path within the theory we want to display, takes a different action. Also note that pretty-printing lemmas is not supported, as it is never needed by the interface - we always display a path within a lemma, never the lemma itself. As one example, let us look at how the theory rules are pretty-printed.

```haskell
build
```

Build the HTML document showing the rules of the theory.

```haskell
rulesSnippet :: HtmlDocument d => TheoryPath thy path = vcat
```

theRules = get Classified Rules thy

rules = get Classified Rules thy

where
rules = get Classified Rules thy

-- Render the image corresponding to the given theory path
pngThyPath :: FilePath -> (Sequent -> D.Dot () ) -> Closed Theory
            -> TheoryPath -> IO FilePath

where
  go (TheoryReqCases i j) = renderDotCode (casesDotCode i j)
  go (TheoryProof l p) = renderDotCode (proofPathDotCode l p)
  go _ = error "Unhandled theory path. This is a bug."

-- Get dot code for required cases
casesDotCode i j = D.showDot $ compact $ snd $ cases !! (i-1) !! (j-1)
cases = map (getDisj . get LrcCases) (getCase Distinctions thy)

-- Get dot code for proof path in lemma
proofPathDotCode lemma proofPath =
  D.showDot $ fromMaybe (return ()) $ do
    subProof <- resolveProofPath thy lemma proofPath
    sequent <- pslInfo $ root subProof
    return $ compact sequent

-- Render a piece of dot code
renderDotCode code = do
  let dotPath = dir <!> getDotPath code
      pngPath = add Extension dotPath "png"
      existsPng <- doesFileExist pngPath
      unless existsPng $ do
        writeFile dotPath code
dummyMsgChan <- newChan
  -- FIXME: Provide better entry point to render Dot files
  graphvizDotToPng "dot" dotPath pngPath dummyMsgChan
  return pngPath

The pngThyPath function is called when a graph is requested from the server. Graphs can only be displayed for certain types of theory paths, other paths don’t have any graphs associated with them. When such a patch is requested, we try to extract the information from the corresponding theory and try to find a pre-rendered graph on disk. If the graph has not already been rendered, we render it using Graphviz and cache the image on disk such that it does not need to be re-rendered again later.

As can be seen, the graphvizDotToPng function at the moment still requires a channel for outputting debug information. We don’t need this information, but have to pass a channel to the function anyway, so we pass it a dummy channel which we throw away afterwards. This should be, if possible, fixed later.
Other modules

Other modules in the server not mentioned before are the Web.Hamlet and Web.Settings modules.

The Web.Hamlet module contains various HTML templates the server uses to display information, for example the template for the front page of the server is included in this module. These templates are mostly static, the pretty-printing of the theories is done in the Web.Theory module. The name of the module stems from the name of the language used for writing the templates, namely Hamlet. It is part of the Yesod web framework.

The Web.Settings module contains various constants used throughout the server code, for example the name of the directory used for caching images or the default port the server should listen on (3001).

4.3 Client/Server interaction

The interaction between the client and the server is done through Asynchronous JavaScript and XML (AJAX) requests. The client can request an action by sending such a request to the server. The server will then reply with a JSON-formatted response, which contains information as to what the client should do next.

On the server side, the response that can be sent is represented using an algebraic data type that can represent all the possible responses that we might want to send to the client.

<table>
<thead>
<tr>
<th>Simple data type for generating JSON responses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>data JsonResponse = HtmlTextContent ^ Title and HTML content</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

On the client side JavaScript code we can find the performASR function which performs an asynchronous request to the server using the ajax function from the JQuery library. The function makes a request to the given URL, requesting the given datatype (usually JSON) and calls the given callback functions on success or failure respectively. The function also sets/unsets the loading screen in the user interface that allows the user to cancel a request if it takes too long.

```javascript
/**
 * Perform an ASR (asynchronous request) to the server.
 * @param path The path to request (absolute!).
 * @param dataType The data to request (xml, html, json).
 * @param cache Should we cache? True/false.
 * @param success The callback function on success.
 * @param error The callback function on error.
 */
performASR: function(path, dataType, cache, success, error) {
  $.ajax({
    url: path,
    dataType: dataType,
    cache: cache,
    success: function(data, textStatus) {
      loadingScreen.reset();
      success(data, textStatus);
    },
    error: function(data, textStatus, err) {
      loadingScreen.reset();
      error(data, textStatus, err);
    }
  });
}
```

After a request, the client can then decode the JSON response easily and perform the appropriate action. For example, if the server told us to redirect to a new page we show the loading screen and...
then redirect. If the server sent us new HTML code, we can perform some arbitrary action with it (mostly we display the new HTML code in the main view).

```javascript
/**
 * Process JSON response from server.
 * @param path The path that was loaded.
 * @param data The data that was received.
 * @param html Callback for html data.
 */
handleJson: function(data, html) {
  // Parse resulting json
  if(data.redirect) {
    // Server wants redirect
    loadingScreen.show(data.redirect);
    window.location.href = data.redirect;
  } else if(data.html) {
    // Server sent back html
    html(data.title, data.html);
  } else if(data.alert) {
    // Server requested alert box
    ui.showDialog(data.alert);
  } else {
    // Server sent unknown request
    ui.showDialog("Received invalid response from server.");
  }
}
```

The functions are as general as possible, and callbacks were used heavily in order to achieve this. The functions above are used throughout the rest of the code in order to perform specific requests, depending on what the user wants to do.

Another feature of the user interface are the keyboard shortcuts. The code below initializes the shortcuts for the keys j, k, J, K and 0–9 using the add_shortcuts function.

```javascript
// Add keyboard shortcuts
var shortcuts = {
  74 : function() { proofScript.jump('next/smart', null); },
  75 : function() { proofScript.jump('prev/smart', null); },
  106 : function() { proofScript.jump('next/normal', null); },
  107 : function() { proofScript.jump('prev/normal', null); }
}
for(i = 1; i < 10; i++) {
  shortcuts[i + 48] = function(key) {
    mainDisplay.applyProofMethod(key - 48);
  };
}
this.add_shortcuts(shortcuts);
```

In the code above, the variable shortcuts is a map which maps keycodes to actions. For example, j corresponds to keycode 106. The shortcuts are then registered with the user interface using the function below.

```javascript
/**
 * Add keyboard shortcut(s) from map.
 * @param map Map of { key : callback } pairs.
 */
add_shortcuts: function(map) {
  $('html').keypress(function(ev) {
    var key = ev.which;
    var tag = ev.target.tagName.toLowerCase();
    // Don't trigger on input/textarea
    if(tag == 'input' || tag == 'textarea') return;
    // If key is in map, call the given
```
In the function `add_shortcuts` we register a new event handler for keypress events on the root of the DOM. If the keypress event was registered on a textarea or input field, we ignore to make sure we don’t interfere with textual input. If that is not the case, we check if the pressed key is in the map of shortcuts and if so hide any possibly open menus and call the callback function corresponding to the shortcut. The callback function is also passed the keycode, which allows us to use the same callback for multiple keys (as is done with the keys 1-9 for proof method application).
5 Case study

5.1 Overview

In this section we perform a case study in order to find out whether incremental security protocol design is a viable idea.

For this purpose we will incrementally verify a key-exchange protocol based on the Needham-Schroeder Public-Key protocol. We first describe the protocol formally. Then we translate the formal description into a theory file that can be understood by the prover and describe all relevant parts. Initially we then verify this description with respect to an external adversary.

Then we attempt verification of the protocol with respect to an internal adversary. This uncovers a flaw in the protocol which we will fix and finally verify that the modified protocol is correct with respect to an internal adversary.

Finally, we attempt verification of the protocol with respect to an internal adversary with session-key compromise capabilities. This uncovers a problem with the description of the security property we try to verify. Again we make a modification and then verify the modified protocol. We then conclude with a section on the lessons learned.

5.2 The key-exchange protocol

5.2.1 Mathematical description

The Needham-Schroeder Public-Key (NS-PK) protocol was proposed by Needham and Schroeder [7]. The following is a description of a simplified key-exchange protocol based on the NS-PK protocol, which we will use for our case study.

Let $pk(X)$ and $sk(X)$ be the respective public and private halves of an encryption key-pair belonging to $X$, with $\{m\}_{pk(X)}$ denoting the public-key encryption of the message $m$ with the public key of $X$ and $m,n$ denoting the concatenation of the messages $m$ and $n$. Also, let $I$ and $R$ be the initiator and responder roles in the protocol. The steps of the protocol are as follows:

1. $I \rightarrow R$: $\{1, mi, I\}_{pk(R)}$
2. $I \leftarrow R$: $\{2, mi, nr\}_{pk(I)}$
3. $I \rightarrow R$: $\{3, nr\}_{pk(R)}$

Let's look at these steps in more detail. First, note that all messages are marked by their order of appearance in the protocol (1, 2, and 3) in order to avoid the possibility of mistaking one message for another. Then in the first step, the initiator sends the initiator nonce and its identity to the responder, encrypted with the public key of the intended recipient. The responder in turn sends back the received nonce and its own responder nonce, encrypted with the public key of the given identity received in the first message. In the last step, the initiator then sends back the nonce (again encrypted). After the exchange, the session key is defined as the hash of the concatenation of both nonces.

5.2.2 Theory file for the prover

Now, let's translate the key-exchange protocol to the notation understood by dh-proto-proof. We will introduce the functions/keywords $\text{encA}$, $\text{lts}$, $\text{anbproto}$, $\text{note}$, $\text{rule}$, $\text{lemma}$, $\text{text}$, $\text{section}$ and $\text{subsection}$ and then use them to construct the theory file.

First, the functions $\text{encA}$ and $\text{lts}$ denote cryptographic primitives. The function $\text{encA}$ refers to asymmetric encryption, with $\text{encA}(m|pk(k))$ denoting the public-key encryption of $m$ with the public key $k$. The function $\text{lts}$ refers to long-term secrets, with $\text{lts}(X)$ denoting the long-term secret associated with $X$. The long-term secret can be used to construct keys, for example $\text{pk}(\text{lts}(X))$.
denotes the public key of X or more specifically the public key half derived from the given long-term secret. Putting both together we can use \( \text{encA}[m] \text{pk}(\text{lts}(X)) \) to denote the public-key encryption of the message \( m \) with the public key of X.

Second, the keyword \( \text{anb-proto} \) is used to define a new protocol using the so-called ,,Alice and Bob” notation. This corresponds to the mathematical notation we used to describe the protocol in section 5.2.1. The keyword \( \text{rule} \) is used to introduce a new rule which can be used as part of the protocol. Internally, and \( \text{anb-proto} \) notation is translated into rules. The \( \text{note} \) keyword can be used within a \( \text{anb-proto} \) section to instruct the prover to keep additional information after a given step. With a \( \text{lemma} \), a security property can be introduced which we want to prove for the given protocol.

Finally, the keywords \( \text{text} \), \( \text{section} \) and \( \text{subsection} \) can be used to introduce structure into the theory file. They are ignored by the prover and can be used to denote sections or add comments to certain parts of the file which are preserved across transformations.

First the preamble: We declare a new theory with the name \( \text{Case_Study_Incremental_Design} \) and declare a new section. Sections are ignored by the theory prover, but can help a human understand the structure of the file.

\begin{verbatim}
theory Case_Study_Incremental_Design
begin
section{* A Key Exchange Protocol based on the NS(L)PK Protocol *}

Next we define the protocol. We use the \( \text{anb-proto} \) notation in order to define the protocol in a somewhat similar way as we have done above, where \( \text{anb-proto} \) stands for ,,Alice and Bob protocol". We put everything in a subsection called ,,Protocol Model".

The definition also includes type assertions, which declare additional typing information to the prover in the otherwise untyped model. There are two general types, fresh values (prefixed ~) and public constants (prefixed with $). Fresh values are values which are drawn from a perfect random number generator, and are guaranteed to be unique. One example of such a value is a nonce. Public constants on the other hand are publicly known constants like the identities of I or R for example.

Finally, the roles also note the session key as being the hash of both nonces, so \( h(ni, nr) \), associated with the given initiator, responder and both nonces of the session.

subsection{* Protocol Model *}

\( \text{anb-proto} \) KEP

\begin{verbatim}
\begin{verbatim}
1. I -> R: \text{encA}[1', ni, I] \text{pk}(\text{lts}(R))
     type assertions:
     ni: "x"
     I : $x$

2. I <- R: \text{encA}[2', ni, nr] \text{pk}(\text{lts}(I))
     type assertions:
     ni: "x"

3. I -> : \text{encA}[3', nr] \text{pk}(\text{lts}(R))
     \text{note} \text{SeKey}(h(ni,nr), <'I', I,R,ni,nr>)
     \text{->} R: \text{encA}[3', nr] \text{pk}(\text{lts}(R))
     \text{note} \text{SeKey}(h(ni,nr), <'R', I,R,ni,nr>)
\end{verbatim}
\end{verbatim}

Next we need to define the adversary model, which we put in the Adversary Model subsection. We define three so-called rules the adversary can make use of, \( \text{Reveal}_{pk} \) which can be used to
reveal a public-key, Reveal_lts which can be used to reveal a long-term secret and SeKey_Reveal which can be used to reveal session keys. For now we want to work with an external adversary only, that is an adversary without the capabilities for compromising long-term secrets and session keys. That is why the last two rules are commented out.

subsection [* Adversary Model *]

rule Reveal_pk:
[ ] --- [ Send( pk(lts($m))) ]

rule Reveal_lts:
[ Knows(m) ] --- [ LTSR(m), Send(lts(m)) ]

rule SeKey_Reveal:
[ SeKey(k, params) ]
---
[ SeKeyReveal(k, params), Send(k) ]

type assertions: k : h(x,y)

In the Security Properties subsection we define the security properties (lemmas) we would like to prove for the protocol. We define one security property with the name SeKey_secrecy which postulates that the session keys derived by the protocol remain secret.

subsection [* Security Properties *]

lemma SeKey_secrecy:
"key :> SeKey(k, <role , l,R,ni,ni>) &
knows :> Knows(k)
==>
(Ex #vr. vr :> LTSR(l))
| (Ex #vr. vr :> LTSR(R))
"

First, let us introduce the operators used in the description of the lemma. The == operator denotes implication, & denotes logical conjunction, and | denotes logical disjunction. With Ex we denote existential quantification. The operator :> denotes that the rule application on the left-hand side introduced the facts on the right-hand side to the state multiset.

The left-hand side of the implication states that there exist two rule applications, referred to as key and knows which introduce the given facts. The key application introduces the custom fact SeKey into the state, which was used to note the session key derived by a run of the protocol. The knows application introduces the fact Knows(k) into the state, meaning that the intruder has gained knowledge of the session-key k. Now if both of those are true, their conjunction and therefore the left-hand side of the implication becomes true.

On the right-hand side, we state state that there must either exist a rule application vr in which the long-term secret of the role l was revealed or the long-term secret of the role R was revealed. The fact LTSR is used to note the revelation of long-term secret in the state of the execution.

To summarise, the lemma states the following: If the protocol was executed (a session key was derived) and the intruder gained knowledge of it, then the long-term secret of either the initiator or the responder must have been revealed. If the lemma does not hold true, this means that it is possible for the intruder to gain knowledge of the session key without revealing the private keys of the participants.
5.3 Defending against an external adversary

After now having constructed a definition of the protocol which can be read by dh-proto-proof, we can now load it and try to automatically prove session key secrecy with respect to an external adversary. We simply run dh-proto-proof interactive working directory where working directory is the directory in which the theory file resides. This will start the server, and the interface can then be accessed through a browser on the local machine. Having started dh-proto-proof, we'll just try to run the autoproof on the session-key secrecy security property and see if the prover can already come up with a proof for our lemma. The result is shown below.

\[
\text{lemma (modulo E) SeKey secrecy:}
\]
\[
\text{"((#key : SeKey( k, <role, l, r, ni, nr])] &}
\]
\[
\text{(#knows : Knows( k ]))) } \Rightarrow
\]
\[
\text{((Ex #vr1 . #vr1 : LTSR( l ))) } \land \text{((Ex #vr2 . #vr2 : LTSR( r )))"}
\]
\[
solve( #knows7 : Knows( k2 ] )
\]
\[
case Knows
\]
\[
solve( #key6 : SeKey( k2, <role5, l, R1, ni3, nr4 ] )
\]
\[
case KEP.I3
\]
\[
solve( #key6 [0 ] < : KEP.I2( $R11, $R12, "ni13", nr4 ] )
\]
\[
case KEP.I2
\]
\[
solve( #knows7 [0 ] < : K( h(<"ni13", nr4 ] ) )
\]
\[
\text{case fake_h}
\]
\[
\text{by solve( #w38 [0 ] < : K( "ni13" ] )}
\]
\[
\text{qed}
\]
\[
\text{qed}
\]
\[
\text{next}
\]
\[
case KEP.R3
\]
\[
solve( #key6 [0 ] < : KEP.R2( $R11, "nr12", l, ni3 ] )
\]
\[
case KEP.R2
\]
\[
solve( #knows7 [0 ] < : K( h(<ni13, "nr12"> ) )
\]
\[
\text{case fake_h}
\]
\[
\text{by solve( #w39 [1 ] < : K("nr12") ] )}
\]
\[
\text{qed}
\]
\[
\text{qed}
\]
\[
\text{Next, we need to know where the fact SeKey( k, <role, l, R, ni, nr] ) could have been derived from. There are two possible cases here: It could have come from the initiator or the responder; i.e. for rule KEP.I.3 or KEP.R.3 in the protocol. This means that the session key could have been leaked by either of the two parties. Now in both cases, the proof investigates the origins of the last step of the given role. In the KEP.I.3 case we try to find its origin by tracing it back to KEP.I.2 (the second step of the initiator role in the protocol) and respectively the same for the responder.}

After having traced the full execution of the protocol from both angles, the autoproof concludes that the only way the adversary could have gained knowledge of the session-key is by having constructed a fake one (case fake_h) and the proof concludes.

5.4 Defending against an active adversary

Now we would like to prove session-key secrecy with respect to an internal adversary. For this we need to change the adversary model in the source file as already mentioned above. This can be easily accomplished by editing the source of the theory within the interactive interface and moving the Reveal_1sts rule outside of the text block. Note that this only works for text blocks, normal single- or multi-line comments are dropped during the parsing process and hence cannot be
reproduced within the interactive web interface. This tripped me up multiple times, and could be improved in a future version.

After having added the Reveal_1 rule through the web interface, the original proof does not hold anymore. After all, we now have an internal adversary. The proof looks like this now:

```plaintext
lemma (modulo E) SeKey_secrecy: 
  "((#key ::> SeKey( k, <role, I, R, ni, nr> )) & 
    #knows ::> Knows( k ))) => 
  ((Ex #vr1. #vr1 ::> LTSR( I ))) | ((Ex #vr2. #vr2 ::> LTSR( R )))"
solve( #knows7 ::> Knows( k2 ) )
case Knows
  solve( #key6 ::> SeKey( k2, <role5, I, R1, ni3, nr4> ) )
case KEP_I3
    solve( #key6 [0] ::< KEP_I2( $111, $R12, *ni3, nr4 ) )
case KEP_I2
      solve( #knows7 [0] ::< K( h(<ni13, nr4>) ) )
case fake_h
        solve( #w38 [0] ::< K( "ni13" ) )
case KEP_I1
          by sorry // not yet proven
        next
case KEP_I3
          by sorry // not yet proven
        next
case KEP_R_2
          by sorry // not yet proven
        qed
      qed
    qed
next
case KEP_R_3
  solve( #key6 [0] ::< KEP_R_2( $R11, *ni13, I, ni3 ) )
case KEP_R_2
  solve( #knows7 [0] ::< K( h(<ni3, nr12>) ) )
case fake_h
    solve( #w39 [1] ::< K( "nr12" ) )
case KEP_I3
      by sorry // not yet proven
    next
case KEP_R_2_case_0
      by sorry // not yet proven
    next
case KEP_R_2_case_1
      by sorry // not yet proven
      qed
    qed
  qed
  qed
next

There are now more case distinctions than we had before, which are by default unproven. But if our original protocol design can also guarantee session-key secrecy with respect to an internal adversary, it should be no problem for the autoprover to find a proof now. So, we can just try to run the autoprover on each sorry step to see if the autoprover can fill in the gaps. This will fill in most of the gaps, but at some point the prover stopped with the message sorry // prover stuck => possible attack found. We'll have to investigate whether this is actually an attack, or whether the prover simply failed to prove the security property even though it still holds.

The prover got stuck in multiple parts of the proof. Let's look at one of the places where the prover got stuck in more detail. Part of the proof is listed again below, please note that I omitted parts of the proof for the sake of brevity. We will focus on the sorry step in the proof below for now.

case Knows
```

solve( #knows7 ::> Knows( k2 ) )
```
solve( #key6 :> SeKeY( k2, <role5, l, R1, ni3, nr4> ) )
case KEP_I_3 // omitted
qed
next
case KEP_R_3
solve( #key6 [0] <: KEP_R_2( $R11, "ni12", l, ni3 ) )
case KEP_R_2
solve( #knows7 [0] <: K( h(<ni3, "ni12">) ) )
case fake_h
solve( #w20 [0] <: K( encA[<1', ni3, l>]pk(lts($R11)) ) )
case KEP_I_1 // omitted
qed
next
case fake_encA
solve( #vr43 [1] <: K( pk(lts($R11)) ) )
case Reveal_pk
solve( #w15 [0] <: K( encA[<3', "ni12">pk(lts($R11)) ) )
case KEP_I_3 // omitted
qed
next
case fake_encA
solve( #w39 [1] <: K( "ni12" ) )
case KEP_I_3
solve( #w72 [0] <: K( encA[<2', "ni67", "ni12">pk(lts($I65)) ) )
case KEP_R_2
solve( #w39 [0] <: K( "ni67" ) )
case KEP_I_1
by sorry // prover stack => possible attack found

If we look at the graph Dh-proto-proof generates when we look more closely at the sorry step where the prover failed (see figure 6 in the appendix) we can see an attack on the protocol. The attack is spelled out below, where \( A(X) \) denotes the adversary acting as a participant in the protocol in the role \( X \).

1. \( I \rightarrow A(R) : \{ ni, I \}_K_{PA} \)
2. \( A(I) \rightarrow R : \{ ni, I \}_K_{PR} \)
3. \( I \leftarrow R : \{ 2, ni, nr \}_K_{PR} \)
4. \( I \rightarrow A(R) : \{ 3, nr \}_K_{PA} \)
5. \( A(I) \rightarrow R : \{ 3, nr \}_K_{PR} \)

This works because the attacker can now pretend to be a participant in the protocol. See the graph mentioned before for a more detailed execution trace (and note that the \( I \) and \( R \) are not synchronized). The attack corresponds to the man-in-the-middle attack on the NS-PK protocol found by Lowe [5], and we can fix it in the same way. We can just modify the second step in our protocol to correspond to the (fixed) Needham-Schroeder-Lowe Public-Key (NSL-PK) protocol:

\[
I \leftarrow R : \{ 2, ni, nr, R \}_K_{PR}
\]

This fixes the synchronization issue and should hopefully also fix the man-in-the-middle attack. We can directly make this modification within the interactive web interface, but this is where I ran into trouble: The prover doesn’t give you the nice and simple anb-proto notation anymore but the parsed, rule-based notation. This doesn’t make it impossible to directly make the modification, but it certainly does make harder to get it right the first time around.

In particular, the source needs to be edited in two places: We need to change the rules KEP_I_2 and the KEP_R_2. In other words, we need to adapt what the responder sends and what the initiator expects to receive (which needs to match up). Below are the rules before the modification.
And now the rules after the modification. Note that when inserting $R$ into the rules a dollar sign needs to be prepended to signify that this is a public constant and not a new, unrelated fresh value. Otherwise this won’t work, which is a possible source of confusion for someone who is new to the utility.

After having made those modifications, much of the proof will become invalid (because the protocol is now somewhat different). These proof steps are nicely highlighted in red though, making it obvious that they don’t apply anymore. Running the autoprove on the full proof again we now get a proof for the security property, meaning session-key secrecy now also hold with respect to an internal adversary in our modified protocol.

5.5 Adding session-key compromise capabilities

Next, we will give the adversary session-key compromise capabilities. This can again be done by interactively editing the theory and uncommenting the last rule, SeKey_Reveal. As as reminder, the rule is reproduced again below.

```
rule SeKey_Reveal:
  [ SeKey(k, params) ]
  -->
  [ SeKeyReveal(k, params), Send(k) ]
type assertions: k: h(x,y)
```

This added two new case distinctions to the full proof of the session-key secrecy security property of the modified protocol, as can be seen below. Both new cases were automatically introduced because the adversary can now make use of the SeKey_Reveal rule, which was not possible before.

```
solve( #key6 [0] <: KEP_I_2( $I11, $R12, "ni13, nr4" ) )
case KEP_I_2
  solve( #knows7 [0] <: K( $<$ni13, nr4$>$) )
  case SeKey_Reveal
  + by sorry // not yet proven
  + next
  case fake_h
  solve( #w24 [0] <: K( $<$encA(<'2', "ni13, nr4, $R12$)>lt$($I11$)) )
```

29
However, trying to automatically prove the missing pieces of the proof will result in failure. This is what should be expected: After all, how can any protocol guarantee session-key secrecy against an adversary which can directly compromise session-keys?

But our original security property does not take this into account. We will need to adapt the security property and make the lemma weaker. As a reminder, this is the original lemma:

\[ \text{Lemma SeKey secrecy:} \]
\[ \begin{align*}
& \text{key} := \text{SeKey}(k, \langle \text{role}, l, R, ni, nr \rangle) \land \\
& \text{knows} := \text{Knows}(k) \\
& \implies \\
& (\text{Ex } \#vr. vr := \text{LTSR}(I)) \\
& \lor (\text{Ex } \#vr. vr := \text{LTSR}(R)) \\
& \lor (\text{Ex } \#vr \text{ rolep. vr := SeKeyReveal}(k, \langle \text{rolep}, l, R, ni, nr \rangle))
\end{align*} \]

We can express the modification we'd like to make thus:

\[ \text{Lemma SeKey secrecy:} \]
\[ \begin{align*}
& \text{key} := \text{SeKey}(k, \langle \text{role}, l, R, ni, nr \rangle) \land \\
& \text{knows} := \text{Knows}(k) \\
& \implies \\
& (\text{Ex } \#vr. vr := \text{LTSR}(I)) \\
& \lor (\text{Ex } \#vr. vr := \text{LTSR}(R)) \\
& \lor (\text{Ex } \#vr \text{ rolep. vr := SeKeyReveal}(k, \langle \text{rolep}, l, R, ni, nr \rangle)) \\
& \lor (\text{Ex } \#vr. vr := \text{SeKeyReveal}(k, \langle \text{role}, l, R, ni, nr \rangle))
\end{align*} \]

We added a new clause to the right-hand side of the implication, stating that is now a third case in which the intruder can gain knowledge of the session-key. That is, the intruder can directly use the rule SeKeyReveal we added to the adversary model in order to reveal the session key.

### 5.6 Lessons learned

I believe that this case study provides evidence for the soundness of the idea of incremental security protocol design. When adapting the protocol, one can automatically see the proof change, with new case distinctions, part of the proof becoming invalid because of protocol changes, and so forth. The automatic prover also works very well, during the case study it was never necessary to manually adapt a proof.

Still, there are some things that could be improved further. Editing the source of a theory requires intimate knowledge of the syntax and can easily go wrong (thankfully, the interactive interface has the ability to undo changes).

Another problem arises when the automatic prover gets stuck and the user is left to decide: Can we see an attack here? Especially with more complex protocols the sequent graphs can become very large, making them hard to navigate and try to understand what is going on. Often the automatic prover also generates larger proofs than could be constructed by hand.

The problems that arose are thankfully not fundamental to the general idea and it seems that incremental security protocol design is certainly something that could be realised in practice with a bit more work.
6 Conclusion

First we presented a user interface for a theory prover implemented as a web application. Then we performed a case study to evaluate the interface and our ideas about incremental security protocol design.

The user interface developed in the thesis significantly improves over the old interface and the associated workflow. Before, the prover only had a static, batch-oriented interface. This restricted the interaction, as the tool had to be re-run after every change in the protocol. With the new interface, the user can dynamically interact with the prover. The protocol rules as well as the security properties can be edited on the fly. Proofs can be constructed by hand, generated automatically, or both. The interface allows for removing parts of proofs, or calling the automatic prover only on certain subproofs. Visualizations for individual proof steps are be generated on the fly and displayed to the user. As a result, constructing proofs has become much more efficient and pleasant.

The second contribution consists of a case study evaluating the user interface and exploring the experimental idea of developing security protocols incrementally along with proofs for their desired properties. The case study shows strong support for the user interface and the ideas put forth. With more time, the idea could certainly be refined and improved upon.

We improved upon the current interface with a new implementation that is significantly better than the old one. We performed a case study to show that this is true. Additionally, we explored and provided support for experimental ideas about security protocol design and showed that they are viable given more time.
7 Bibliography

References


A Additional figures

Figure 6: Attack on the key-exchange protocol