Module 03: Introduction to Tamarin Proofs

Week-7: The Visa contactless protocol

David Basin, Xenia Hofmeier and Sofia Giampietro

basin@inf.ethz.ch, xenia.hofmeier@inf.ethz.ch, sofia.giampietro@inf.ethz.ch

November 2022

Welcome to the lab session for week 7 of the Information Security Lab. This session is part of the Module-03 on *Introduction to Tamarin Proofs*.

Overview

EMV, founded by companies Europay, Mastercard, and Visa (thus the name) is the international protocol standard for in-store smartcard payments. In this lab we will develop a Tamarin security model and use it to perform a formal analysis of a simplified version of the EMV protocol for *Visa contactless transactions*.

The goal of this lab is for you to deepen your modeling and analysis skills, with the use case of a real-world protocol that is used in over 2 billion payment cards and 40 million payment terminals worldwide. Specifically, you will learn how to model (i) cryptographic mechanisms involving both asymmetric and symmetric crypto, (ii) communication channels with different levels of security, (iii) multiple roles, and (iv) branching execution (similar to *if* statements in imperative languages).

Please note that we expect you to use the Tamarin tool. We will **not** accept submissions written in any other languages or tools, including manually developed proofs.

Tips when using Tamarin

- The lab exercises have been carefully selected so that Tamarin terminates quickly. If Tamarin is taking more than a couple of minutes to finish, there is likely a problem with your model.
- Similarly, the lab exercises do not require an excessive amount of RAM and if you see that Tamarin is consuming your system resources, it is likely there is a problem with your model.
- You can use Tamarin to check that a `spthy` file is well formed by passing it to Tamarin without arguments, e.g. `tamarin-prover UnsignedDH.spthy`. Tamarin should report that all wellformedness checks were successful.
- It is vital that the sanity lemmas are proved successfully. If they fail, you may

be proving theorems about the empty set of traces, where all security properties are vacuously true.

- When investigating an attack or debugging a model, it can be helpful to use Tamarin's interactive mode to step through the model.
- The Tamarin Manual contains a lot of helpful advice, so consulting it is highly recommended. https://tamarin-prover.github.io/manual/

Overview of EMV contactless payment

Cashless payment methods have become very popular in the last few years, specially in the context of the recent Covid-19 pandemic. These methods typically use devices like smartphones or smartwatches to transact with the payment terminals via Near Field Communication (NFC), a technology that allows for short-range wireless communication. In transactions using NFC, which are called *contactless*, the payment card (or the payment device) is held near the payment terminal and hence the two parties exchange messages called *commands* and *responses*. In this lab we'll regard them simply as messages.

In the EMV contactless transaction, after the card-terminal interaction is completed, the terminal continues with one of three options:

1. Decline offline,
2. Accept offline, or
3. Request Online Authorization.

If the transaction is accepted offline (i.e. second case), which occurs in some cases where the transaction amount is small or Internet connectivity is limited, the funds are *not* immediately transferred from the cardholder's account to the merchant's, but at the end of the day or even within a few days. If the transaction is otherwise required to be authorized by the card issuer (i.e. third case), which is the most common scenario, then the terminal goes online, establishes a communication channel with the issuer, and sends the authorization request. Upon reception of the request, the card issuer undergoes a number of checks, such as that the cardholder's account has sufficient funds for the amount requested and that the card-produced cryptographic proof for the transaction is correct. If everything checks out, the issuer sends back to the terminal the authorization response. See below a graphical representation of the interaction between the card, the terminal, and the issuer in a contactless transaction.

The card-terminal interaction is determined by one of the six EMV contactless protocols (called *kernels* in EMV's terminology), each of them corresponds to a different card brand: Mastercard, Visa, American Express, JCB, Discover, and UnionPay. As said before, we will focus on a simplified variation of the Visa protocol.

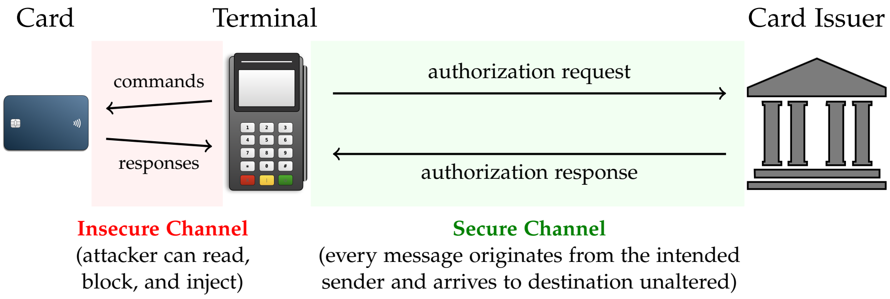
Card
Terminal
Card Issuer
commands
responses
authorization request
authorization response
Insecure Channel
(attacker can read, block, and inject)
Secure Channel
(every message originates from the intended sender and arrives to destination unaltered)

Figure 1: The three agents and their interaction during a contactless transaction.

The specifications of each of these kernels are available at `https://www.emvco.com`, but you do **not** need to read them for this lab!

The (simplified) Visa protocol

This section provides technical background on the specific Visa contactless protocol. For the purpose of this lab, some messages have been simplified and substantial parts of the protocol have been omitted, such as cardholder verification. An overview of the resulting execution flow of the Visa contactless transaction is given in Figure 2. The crypto-related notation in the figure and also in the rest of this material is the following:

- mk is a (master) key only known to the card and its issuer.
- f is a key derivation function.
- $(privC, pubC)$ is the private/public key pair of the card. $pubC$ is publicly known.
- $sign_{priv}(m)$ is the digital signature on m with the private key $priv$. Note that we consider signatures that do not reveal the message m.
- $verify(sig, m, pub)$ equals true if and only if $sig = sign_{priv}(m)$ and $(priv, pub)$ is a valid private/public key pair.
- $MAC_k(m)$ and $MAC'_k(m)$ are Message Authentication Codes (MAC) on m with the key k.

The remainder of this section describes in detail the steps of the protocol.

1. *Application Selection*: An EMV contactless transaction is performed using one of the six EMV contactless protocols/kernels mentioned before. The negotiation of the kernel to be used for the transaction is the first phase of the protocol. This negotiation is done via two exchanges of '`SELECT`' commands and responses.

   This brief description of the application selection phase is merely informative. We will **not** model this phase.

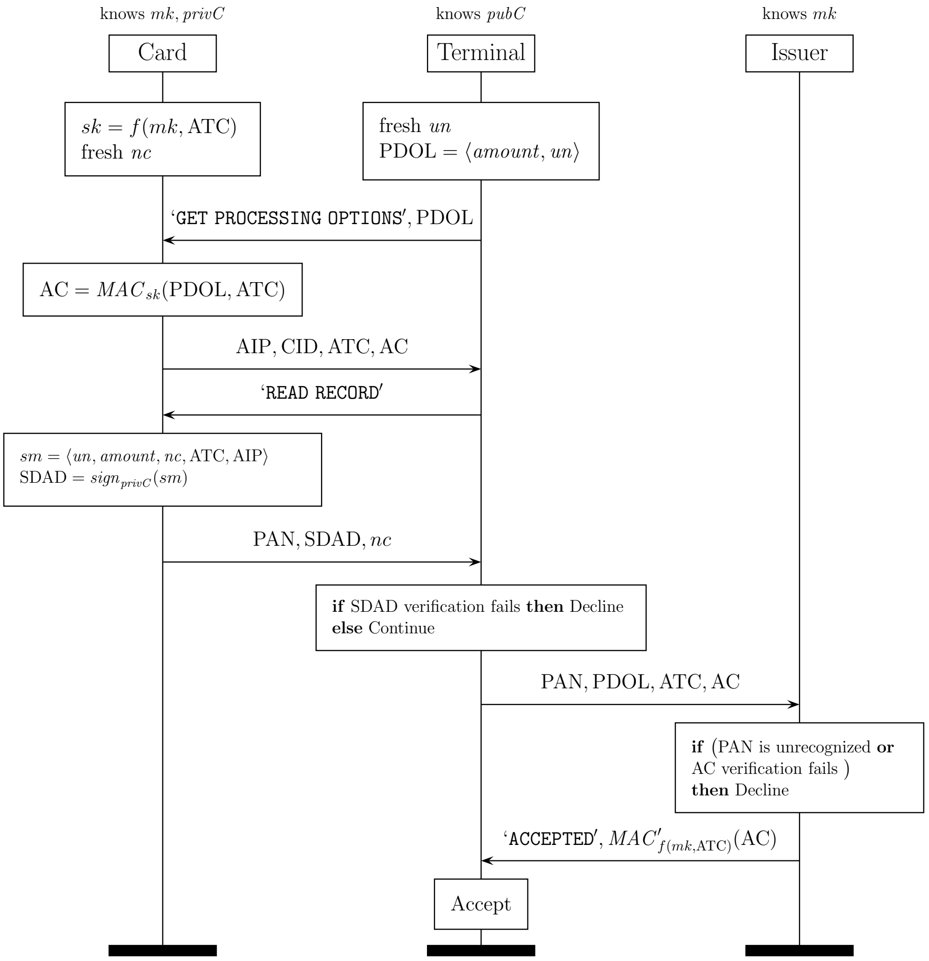
knows $mk, privC$
Card
knows $pubC$
Terminal
knows mk
Issuer
$sk = f(mk, \text{ATC})$
fresh nc
fresh un
PDOL $= \langle amount, un \rangle$
'GET PROCESSING OPTIONS', PDOL
AC $= MAC_{sk}(\text{PDOL}, \text{ATC})$
AIP, CID, ATC, AC
'READ RECORD'
$sm = \langle un, amount, nc, \text{ATC}, \text{AIP} \rangle$
SDAD $= sign_{privC}(sm)$
PAN, SDAD, nc
if SDAD verification fails **then** Decline
else Continue
PAN, PDOL, ATC, AC
if (PAN is unrecognized **or**
AC verification fails)
then Decline
'ACCEPTED', $MAC'_{f(mk,\text{ATC})}(\text{AC})$
Accept

Figure 2: Overview of the (simplified) Visa contactless transaction. While in the full protocol further actions and messages follow the Accept and Decline actions, for this lab we omit them for simplicity.

2. *Offline Data Authentication (ODA)*: After a kernel has been selected (Visa in our case), the terminal sends the 'GET PROCESSING OPTIONS' command with the Processing Data Object List (PDOL) as the payload. The PDOL is composed of the *amount* of the transaction and a terminal-generated random number *un*, called the Unpredictable Number (UN) in EMV's terminology.

   The card responds to this command with the Application Interchange Profile (AIP) and some other data that we will explain later in the Transaction Authorization phase. The AIP informs the terminal of the card's authentication capabilities. In this lab we will assume that AIP = 'fDDA' and will explain later what this means.

   The terminal then sends the 'READ RECORD' command to read the card's records, which are static data stored in the card. We will assume these records are composed solely of the Primary Account Number (PAN, commonly known as the *card number*) and the nonce *nc*.

   The terminal then proceeds to cryptographically authenticate the card, using the method indicated by the AIP. Our assumption on the AIP being 'fDDA' entails that the Fast Dynamic Data Authentication (fDDA) method will be used. In this method, the card replies to the terminal's 'READ RECORD' command with the Signed Dynamic Authentication Data (SDAD), which is a signature by the card on transaction data as indicated in the figure.

3. *Transaction Authorization (TA)*: For transaction authorization, the card generates and supplies the Application Cryptogram (AC) in response to the ‘GET PROCESSING OPTIONS′ command (note that this command goes before the ‘READ RECORD′ one). The AC is a cryptographic proof of the transaction. The computation by the card (and verification by the issuer) of the AC uses the session key *sk* derived from a shared master key *mk* and the ATC. The key *mk* is only known to the issuer and the card.

   Together with the AC, the card sends the AIP (which we described before), the Cryptogram Information Data (CID), and the Application Transaction Counter (ATC). The CID, which can be ‘TC′ or ‘ARQC′, indicates the type of authorization: offline or online, respectively.

   The transaction continues as follows:

   - if CID = ‘TC′, then the transaction is approved offline by the terminal. In this case, the AC is called the Transaction Cryptogram (TC).
   - otherwise (i.e. if CID = ‘ARQC′) the terminal forwards the transaction data to the issuer for online authorization. The data forwarded is composed of the card number (i.e. the PAN), the PDOL, the transaction counter ATC, and the Application Cryptogram (AC). In this case, the AC is called the Authorization Request Cryptogram (ARQC).

     The issuer authorizes the transaction if the AC is correct, in which case it sends back to the terminal the authorization code ‘ACCEPT′ and an issuer-generated MAC, called the Authorization Response Cryptogram (ARPC). The ARPC serves, among other things, as a cryptographic proof for the terminal that the issuer accepted the transaction.

Notice that the terminal **can neither** verify the AC nor the ARPC because the key *mk*, and *sk* by extension, are unknown to it.

Tasks

Download the complementary material from:

https://moodle-app2.let.ethz.ch/mod/resource/view.php?id=819745

which contains the file:

- Visa.spthy, which is a skeleton model for the (simplified) Visa contactless protocol we just described.

Task. Complete the skeleton model by *adding* the appropriate Tamarin code (e.g. rules, lemmas, restrictions) to it such that the resulting model satisfies the following requirements. The only rule you may modify is Card_Responds_To_GPO.

1. It must model the three roles present in a transaction: the *card*, the *terminal*, and the *issuer*.

2. It must allow for the execution of the two types of accepted transactions: *offline* and *online*.

3. Every rule where the *card completes a transaction* must have the form:

   ```
   rule Name:
     [ ... ]
   --[ Running(~PAN, ’Terminal’, <’Card’, ’Terminal’, transaction>),
       Running(~PAN, $Issuer, <’Card’, ’Issuer’, transaction>), ... ]->
     [ ... ]
   ```

4. Every rule where the *terminal completes its interaction with the card for a transaction requiring online authorization* must have the form:

   ```
   rule Name:
     [ ... ]
   --[ Running($Terminal, $Issuer, <’Terminal’, ’Issuer’, transaction>), ... ]->
     [ ... ]
   ```

5. Every rule where the *terminal accepts a transaction offline* must have the form:

   ```
   rule Name:
     [ ... ]
   --[ Commit(’Terminal’, ~PAN, <’Card’, ’Terminal’, transaction>), ... ]->
     [ ... ]
   ```

6. Every rule where the *terminal accepts an online-authorized transaction* must have the form:

   ```
   rule Name:
     [ ... ]
   --[ Online(),
       Commit(’Terminal’, ~PAN, <’Card’, ’Terminal’, transaction>),
       Commit($Terminal, $Issuer, <’Issuer’, ’Terminal’, transaction>), ... ]->
     [ ... ]
   ```

7. The `Commit` and `Running` facts in the above rules must be used for the agreement-based authentication properties, which will be described later.

8. The `~PAN`, `$Terminal` and `$Issuer` terms in the above rules must be the identities of the agents instantiating the card, terminal, and issuer roles, respectively. The term `transaction` must be a message composed of the PAN, the PDOL, the ATC, and the AC.

9. The *card-terminal channel* must be modeled as being under the full control of the adversary who can read, block and inject messages (recall that this is kwnon as the Dolev-Yao adversary model). For this you must use Tamarin's `In` and `Out` facts.

10. There are no rules to model the *terminal-issuer channel*, the *compromise of agents*, or the *issuer role*, other than the ones provided in the skeleton model.

11. The `executable_...` lemmas provided in the skeleton model must be verified. They are sanity check lemmas to verify that your model is operable, i.e. it admits the completion of "clean" transactions without adversarial intervention other than the delivery of messages sent over the card-terminal channel.

12. It must allow for the analysis of the three authentication properties:

    (a) All transactions accepted offline by the terminal are authenticated to it by the card. The lemma for this must be named:
    `auth_to_terminal_offline`.

    (b) All online-authorized transactions accepted by the terminal are authenticated to it by *both* the card and the issuer. The lemma for this must be named:
    `auth_to_terminal_online`.

    (c) All transactions accepted by the issuer are authenticated to it by *both* the card and the terminal. The lemma for this must be named:
    `auth_to_issuer`.

13. The three authentication properties must be implemented using the **non-injective agreement** property as defined in https://tamarin-prover.github.io/manual/book/007_property-specification.html with the message to agree on being $\langle \text{PAN}, \text{PDOL}, \text{ATC}, \text{AC} \rangle$, as explained before in item 8.

Evaluation

You must provide the resulting Tamarin `.spthy` file. As in the previous lab, please name your model file `Visa_<leginumber>.spthy`, where `<leginumber>` is your matriculation (legi) number, *without* dashes.

Tamarin must be able to process the file and prove or disprove each of the properties automatically, i. e. by running

```
tamarin-prover --prove <model>.spthy
```

Make sure that Tamarin does not raise warnings about wellformedness checks failing.

Please submit this on Moodle, adding it to the files you have submitted in last week's lab:

https://moodle-app2.let.ethz.ch/mod/assign/view.php?id=782460

As usual, you may submit as many times as you want.

Hints

Below we give a few hints that you might find be useful when developing your model:

1. The skeleton model offers the rule `Generate_ATC`, which models the "pool" of the Application Transaction Counters (ATC). The card looks the counter up

in this pool in every transaction. Notice that the `Once(...)` facts of the provided `Card_Responds_To_GPO` and `Issuer_Receives_AC` rules along with the `once` restriction make sure that the ATC is different per session, from a card's perspective (note that two different cards can have the same ATC, which is a realistic scenario). You should not need further code to model the generation or lookup of the ATC as your card rules can carry it in the state facts across the full transaction.

2. To use the terminal-issuer secure channel provided in the skeleton model (see rule `Terminal_Issuer_Channel`), you should instantiate the `channelID` term within the protocol rules such that it is compatible with the Issuer rules. Have a look at the `Send` and `Recv` facts in the issuer rules: the `channelID` is of the form `<id, label>` where `id` is a session identifier and `label` is the number of the message in said session.

3. To deal with execution branching in Tamarin, it is a common practice to "multiply" rules. For example, say you have a protocol where an agent goes to two different states, depending on what it receives. This can be modeled with the two rules:

   ```
   rule Go_To_1:
     [ State_0(), In(’go_to_1’) ]-->[ State_1() ]

   rule Go_To_2:
     [ State_0(), In(’go_to_2’) ]-->[ State_2() ]
   ```

   You might use this idea to model the **if** statement for offline vs. online authorization.

4. Finally, decline actions need not be modeled explicitly (at least not in this lab). For example, say you are modeling an agent that stops the execution, provided that some condition *C* is met. Then you should simply have a rule that models the continuation of the execution, provided that the **negation** of *C* holds. See this idea implemented in the provided `Issuer_Receives_AC` rule where the issuer checks that the AC is correct and that the PAN indicates one of its cards. This is indeed the negation of the condition for the issuer's decline action in Figure 2.

5. You might have noticed that the definition of non-injective agreement in the Tamarin manual is very similar to the agreement properties we used in the first lab. The agreement property defined in https://tamarin-prover.github.io/manual/book/007_property-specification.html is as follows:

   ```
   lemma noninjective_agreement:
   "All a b t #i.
       Commit(a,b,t) @i
       ==> (Ex #j. Running(b,a,t) @j)
           | (Ex C #r. Reveal(C) @ r & Honest(C) @ i)"
   ```

   You might also have noticed that this lemma is however a bit different from the agreement lemmas used in the first lab: the above definition makes use of `Honest` facts. In particular, the last line states that the property only has to hold if the agents that were declared `Honest` at the same time point as the `Commit` fact are not compromised (i.e. there is no `Reveal` fact for them).

In most cases (as in the models of the first lab), the only agents that we require to be honest are those who are involved in the `Commit` facs. In these cases we can hence formulate this property without using `Honest` facts. In particular if we assume that there is only a `Honest(a)` and `Honest(b)` fact at `@i`, this lemma can be reformulated as follows:

```
lemma noninjective_agreement:
 "All a b t #i.
   Commit(a,b,t)@i
   & not (Ex #r. Reveal(a)@r) & not (Ex #r. Reveal(b)@r)
   ==>  (Ex id #j. Running(b,id) @ j)"
```

These are equivalent, and in case you are using the second formalism, you can remove the `Honest` facts from the skeleton to prevent wellformedness errors.

Good luck!

Acronyms

AC Application Cryptogram. 5, 6, 8

AIP Application Interchange Profile. 4, 5

ARPC Authorization Response Cryptogram. 5

ARQC Authorization Request Cryptogram. 5

ATC Application Transaction Counter. 5–8

CID Cryptogram Information Data. 5

fDDA Fast Dynamic Data Authentication. 4

MAC Message Authentication Code. 3, 5

NFC Near Field Communication. 2

ODA Offline Data Authentication. 4

PAN Primary Account Number. 4–6

PDOL Processing Data Object List. 4–6

SDAD Signed Dynamic Authentication Data. 4

TA Transaction Authorization. 5

TC Transaction Cryptogram. 5

UN Unpredictable Number. 4