
Une école de l'IMT

# TTool

`ttool.telecom-paristech.fr`

## SysML-Sec Tutorial

| | **Document Manager** | **Contributors** | **Checked by** |
|---|---|---|---|
| **Name** | Ludovic APVRILLE | Ludovic APVRILLE | Ludovic APVRILLE |
| **Contact** | ludovic.apvrille@telecom-paristech.fr | | |
| **Date** | November 6, 2024 | | |

# Contents

# 1 Preface

## 1.1 Table of Versions

| Version | Date | Description & Rationale of Modifications | Sections Modified |
|---|---|---|---|
| 1.0 | April 3rd, 2018 | First draft | |

## 1.2 Table of References and Applicable Documents

| Reference | Title & Edition | Author or Editor | Year |
|---|---|---|---|
| | | | |

## 1.3 Acronyms and glossary

| Term | Description |
|---|---|
| | |

## 1.4 Summary

This document describes how to use SysML-Sec using simple examples[1]. In particular, it covers requirements, attack trees, HW/SW partitioning and software design.

[1]This document has been started in the scope of the AQUAS european project

# 2 Configuration

## 2.1 TTool configuration

At first, if not already configured[2], you must open the configuration file of TTool. The default file is located in:

`TTool/bin/config.xml`

Open your configuration file, and add the following lines accordingly with your TTool and ProVerif installation:

- Directory in which formal specifications for security proofs are generated:

  ```
  <ProVerifCodeDirectory data="../proverif/" />
  ```

- Path to the proverif executable file:

  ```
  <ProVerifVerifierPath data="/opt/proverif/proverif" />
  ```

- Host on which the proof will be started (for example, you could execute this proof on a dedicated machine if the "ProVerifCodeDirectory" is reachable from that dedicated machine):

  ```
  <ProVerifVerifierHost data="localhost" />
  ```

## 2.2 External tools

The configuration for the DIPLODOCUS simulator assumes that a **C compiler**, referenced by the provided Makefile (default = "gcc"[3]) is installed on your machine, as well as the **POSIX-1 librairies**. Also, a Makefile utility must be installed (e.g., "GNU make"[4]).

[2] Your version of TTool should be already configured

[3] `https://gcc.gnu.org/`

[4] `https://www.gnu.org/software/make/`

# 3 Getting started with a toy example

This very first example explains how to use the main capabilities of SysML-Sec.

## 3.1 Getting the example

Be sure to get the latest version of TTool including the remote loading of models (March 2018 and after). Do: File, Open from TTool repository, and select "SysMLSecTutorial.xml".

## 3.2 Understanding the model

The first tab of the model presents an overview of the SysML-Sec methodology (see Figure 1). Each stage of the method is represented with a rectangle that contains a link to the corresponding diagrams. All other tabs correspond to the diagrams of the model.

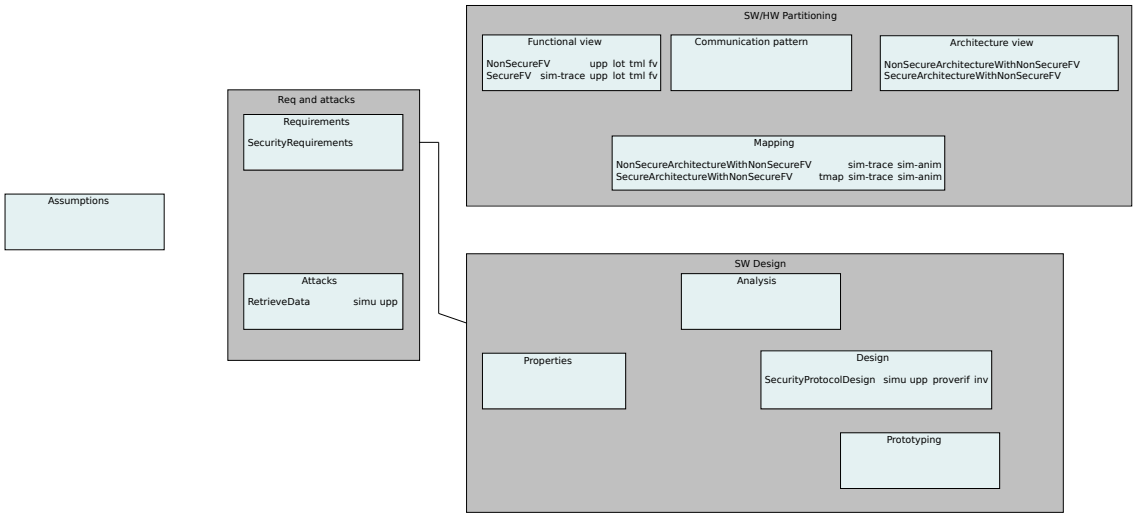


Figure 1: The first diagram represents the SysML-Sec method. Each stage of the method is represented by a rectangle that contains a link to all diagrams of the corresponding stage.

## 3.3 Security requirements

Security requirements are captured with a SysML requirement diagram that is extended in the following ways:

1. Requirements can be tagged as "Functional", "Non Functional", "Privacy", "Confidentiality", "Non Repudiation", "Controlled Access", "Availability", "Immunity", "Data Origin Authenticity", "Freshness", "Business", "Stakeholder Need", "Other"

2. Requirements have a **risk** attribute (low, medium, high).

3. References to other diagram elements can be added and linked to requirements with a "satisfy" relation. "elt satifies req" means that elt is part of the mechanisms used to satisfy the referenced requirement.

The requirement diagram of Figure 2 shows a confidentiality requirements that states that all functional communication paths should be confidential.

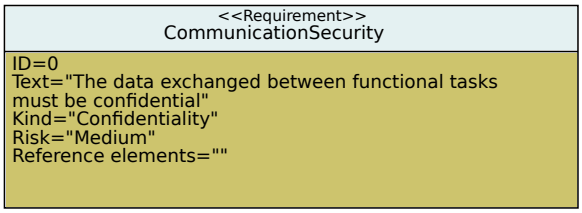


Figure 2: Security requirements (version #1)

## 3.4 Functional Model (version 1)

The functional model is built upon the merge of a SysML Block Definition Diagram and a SysML Internal Block Diagram, see Figure 3. The functional view contains two blocks : T1 and T2. The functional behavior of T1 and T2 is captured with Activity diagrams, as displayed on the left and right of Figure 3. Basically, T1 writes one data sample, and T2 reads one data sample.

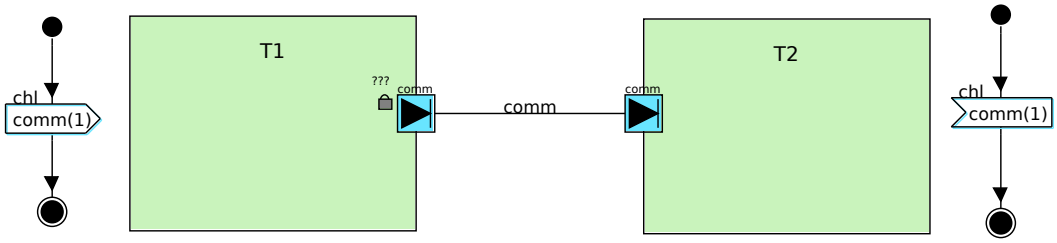


Figure 3: Functional View (version #1)

As shown on the block diagram, T1 and T2 are connected with a data communication channel. Since this communication channel must be secure (see Figure 2), we can now enrich the Requirement Diagram with a new security requirement connected to the initial security requirement (see Figure 4)

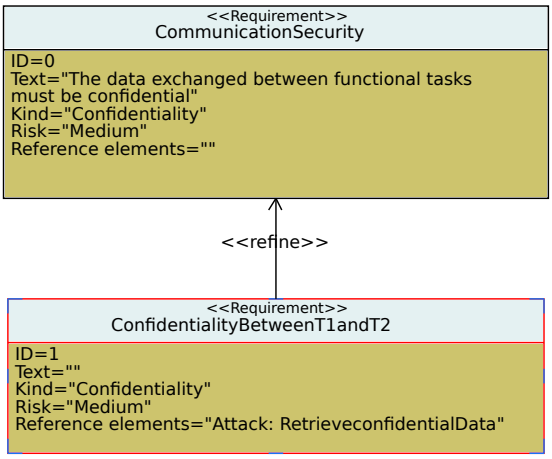


Figure 4: Security requirements (version #2)

We also enrich the communication channel *comm* between T1 and T2 with a confidentiality property, as shown by the *grey lock* with the question marks just next to the port of comm in T1 (Figure 3).

## 3.5 Architecture and Mapping Models (version 1)

A simple architecture model and mapping is shown in Figure 5. T1 and T2 are mapped on CPU1 and CPU2, respectively. The channel between T1 and T2 is mapped on "MainBus" and on "MainMemory".

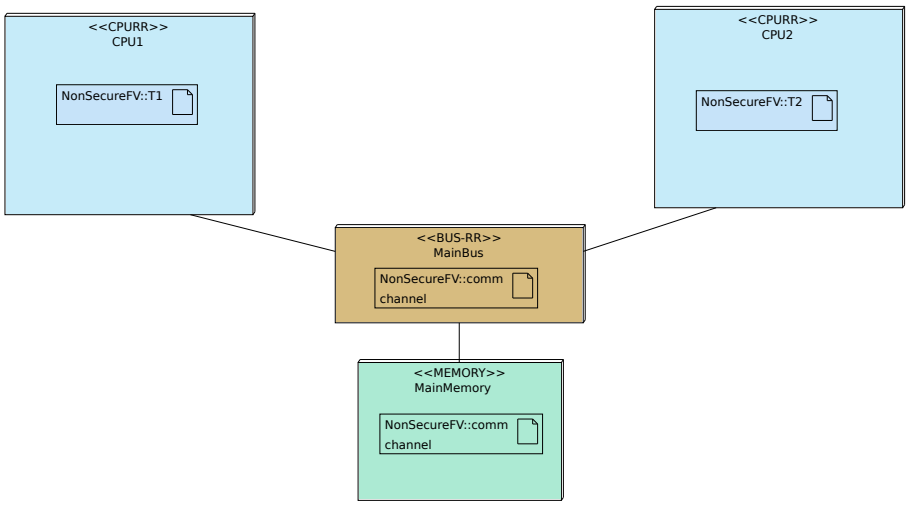


Figure 5: Mapping Model (version #1)

## 3.6 Attack Tree Model

We can now capture potential attacks on this system in an attack tree model. However, SysML doesn't propose any way to capture fault or attack trees. TTool thus proposes relying on SysML parametric diagrams in order to capture attacks (or faults).

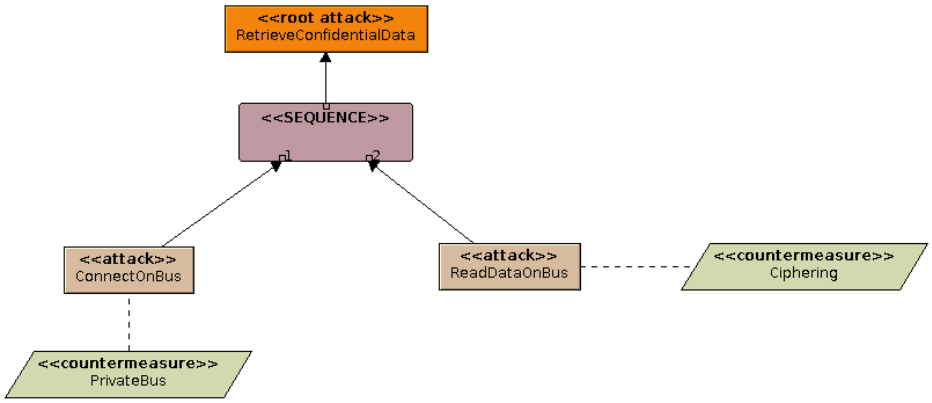


Figure 6: Attack Tree Model

The attack tree of Figure 6 contains a root attack: "RetrieveConfidentialData". This root attack is possible if and only if the attacker first connects to the bus (let's call this attack $att_1$), and then reads (and interprets) a data on the bus ($att_2$). Making either $att_1$ or $att_2$ not possible in our system would be sufficient to ensure that the root attack is not possible. This is obvious in our attack tree, but this is not always the case, such as for larger systems with complex logical combinations of attack steps. Thus, TTool proposes a way to investigate if a given attack is still possible in the system directly from attack trees. Let's try together:

1. Right click on the root attack, and click on "Select for Reachability/Liveness".

2. Let's now see if the root attack is reachable. For this, check the syntax of the diagram, and click on the "Safety verification (internal tool)" icon. Then, select "Reachability of selected states" and click on start. Close the window. You diagram should be annotated with a green "R", as shown on Figure 7.

3. Let's make $att_1$ or $att_2$ not feasible. For this, you can right click on e.g. $att_1$ and select "Disable". If an attack is disabled, it probably means that a countermeasure has been used. Thus, countermeasure blocks can be linked to attacks. In the case of $att_1$, the countermeasure is to make the bus private (e.g., make it internal to a chip if the attacker has no way to investigate a bus within a chip). For $att_2$, a common countermeasure is to use security protocols relying on ciphering algorithms. For now, we assume that only $att_1$ is disabled. Run the verification process again. After this verification process, you should obtain a red "R", meaning that the root attack is no longer feasible (see Figure 8)

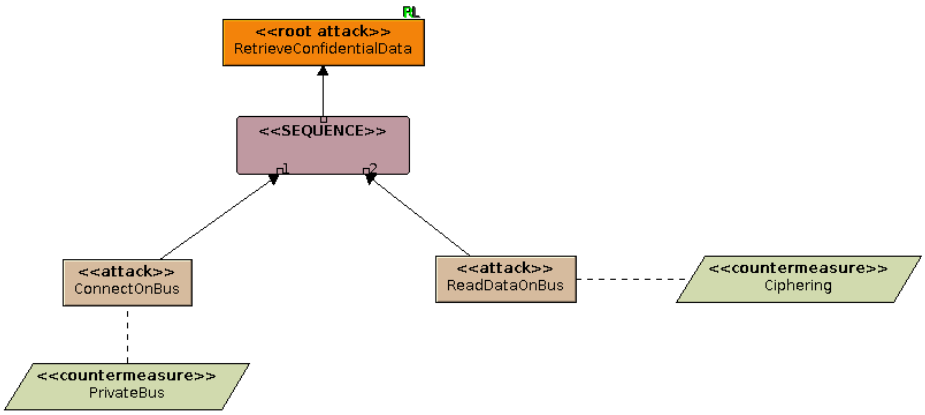


Figure 7: Attack Tree Model after formal verification with no attack steps disabled

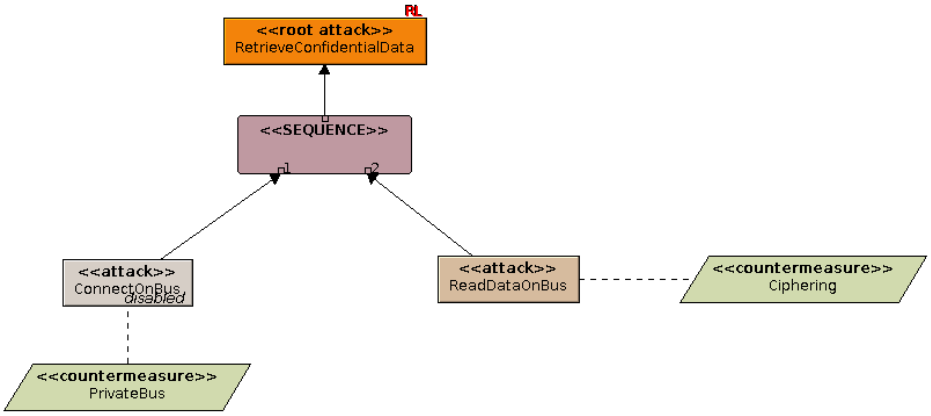


Figure 8: Attack Tree Model after formal verification with attack steps disabled by countermeasures

## 3.7 Implementing security countermeasures

### 3.7.1 Security verification

Before implementing security countermeasures in the mapping model, this document explains how to perform formal security verification. Again, Figure 3 contains a security property on channel "comm" to verify that the latter is confidential. Let's now prove this confidentiality property on the first system version (see Figure 5):

1. Check the syntax of "NonSecureArchitectureWithNonSecureFV" diagram.
2. Click on the "SecurityVerification (ProVerif)" icon. A dialog window should open: click on start. The results should be as shown in Figure 9.
3. The result of the verification is displayed in the lower part of Figure 9. There are two results:
   - The *comm* channel is NOT confidential, which proves that the confidentiality requirement is NOT satisfied
   - The read and write actions in T2 and T1 are reachable. This result is important in the case the property is satisfied. Indeed, if none of these actions were reachable, the channel would be confidential since there would be no exchange of data on this channel.
4. Since *comm* is not confidential, TTool can draw an attack trace that shows how the attacker manages to retrieve the data. Right click on the non satisfied authenticity, and select "show trace" (see Figure 10). This (obvious) trace explains that T1 directly send a data to the attacker since T1 writes the data on a public bus.
5. The functional view is annotated with the verification results, as shown on Figure 11.

### 3.7.2 Countermeasure 1: Secure Bus

As seen before, a first countermeasure is to use a secure bus, which is called "private" in TTool. Thus, the bus in the second mapping (called "SecureArchitectureWithNonSecureFV") is private. You can see this with the green shield icon on the bus. A double-click on this bus makes it possible to change this parameter (public, private).

Try to run the security verification of this second system. You should observe that the confidentiality property is now verified.

### 3.7.3 Countermeasure 2: Secure Functions

A second countermeasure consists in adding security mechanisms to the two functions T1 and T2. TTool offers cryptographic configurations to add security mechanisms to the behaviour of blocks (See our Modelsward 2017 paper). Basically, a cryptographic configuration specifies the type of security mechanism (symmetric cipher, hash, key manipulation, nonce, etc.) and its performance impact in terms of complexity operations by sample.

The modified activity diagrams of T1 and T2 are given in Figure 13. Note that only the activity diagrams have been modified with regards to previous version.

If you double-click on the SE operator of T1, the following windows should open (see Figure 14). This dialog window contains the following fields:

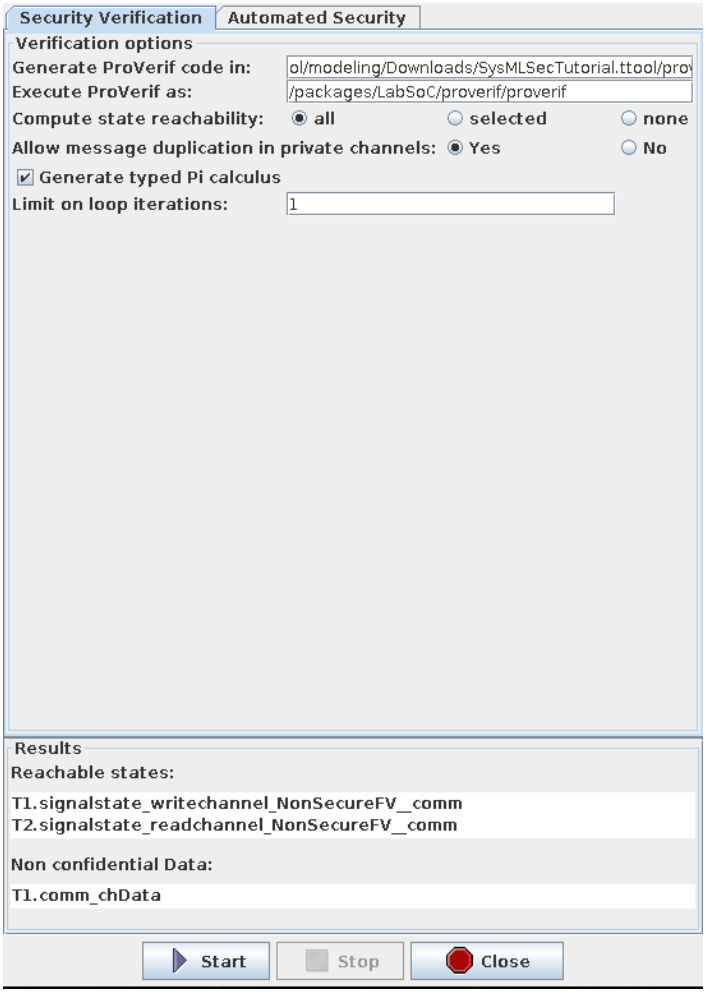


Figure 9: Security verification dialog window

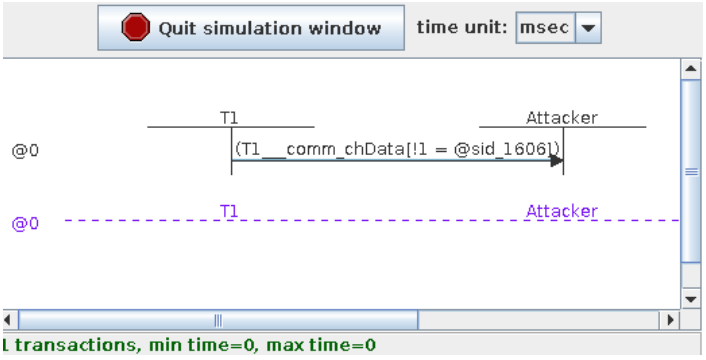


Figure 10: Attack trace

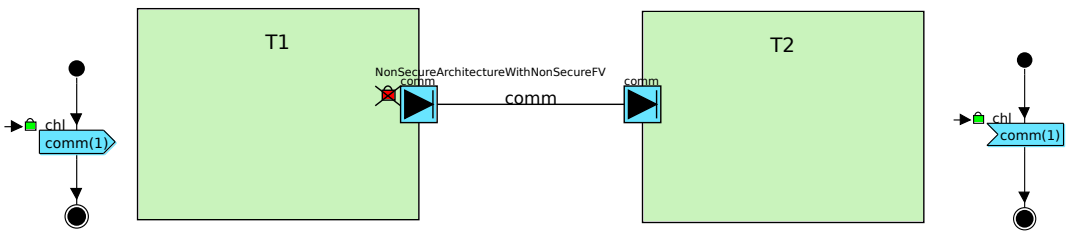


Figure 11: Functional View annotated with security verification

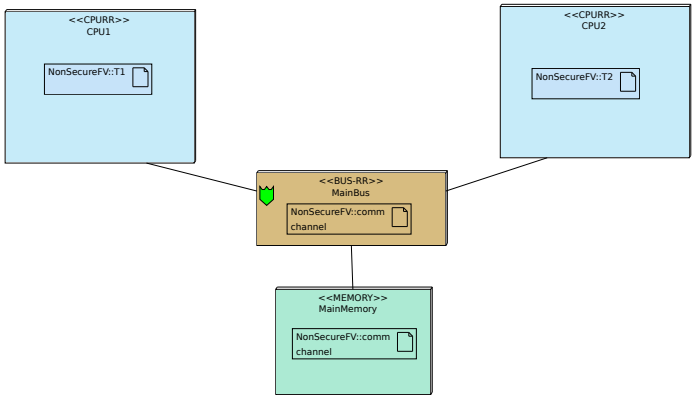


Figure 12: Secure Architecture due to a private bus

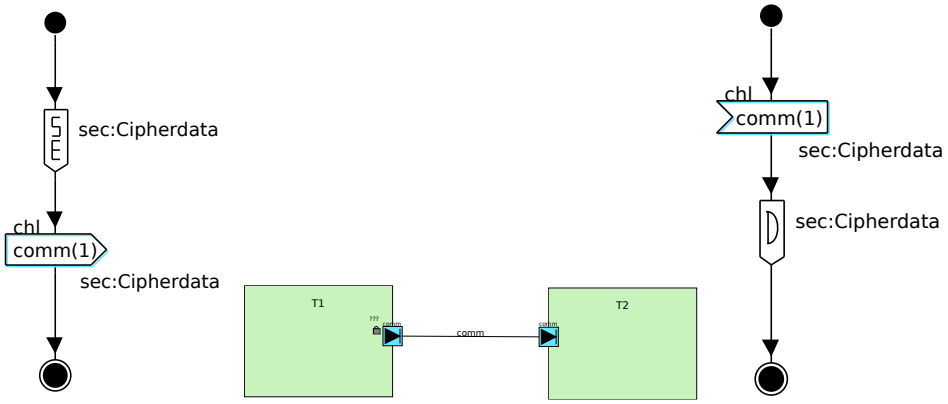


Figure 13: Functional view updated with cryptographic configurations

- **Name** of the configuration. This name is useful to reference a given cryptographic configuration when writing/reading data. For instance, the write operator on *comm* in T1 uses the Cipherdata cryptographic configuration.
- The **type** of the cryptographic configuration: Symmetric, Asymmetric, MAC, Hash, Nonce, Advanced. In our case, a symmetric encryption is selected.
- The **complexity**, in terms of integer operations, of the selected operation
- The use of **cryptographic material**: keys, nonces and precise algorithm (AES, etc.)

Channels can be tagged with cryptographic configurations on the Security tab, with the options shown in Figure 15. By default, channels tagged with cryptographic configurations will send the data in their encrypted form. However, with the use of Hardware Security Modules described below, it may be necessary to send the unencrypted data to the HSM to encrypt, where 'Encrypted form' needs to be marked 'No'. In addition, if a task is to be an attacker task (see our paper in Modelsward 2018), channels can also be marked as 'attacker channels'. If a channel sends data to be secured in the unencrypted form, on the activity diagram, the Cryptographic Configuration name is displayed in red.

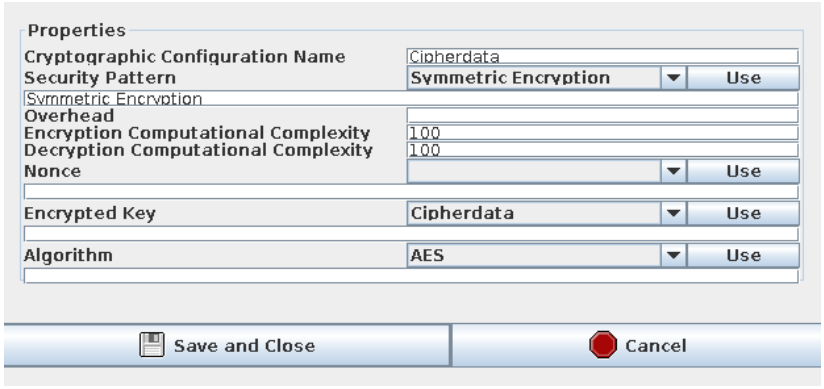


Figure 14: Cryptographic configuration dialog window

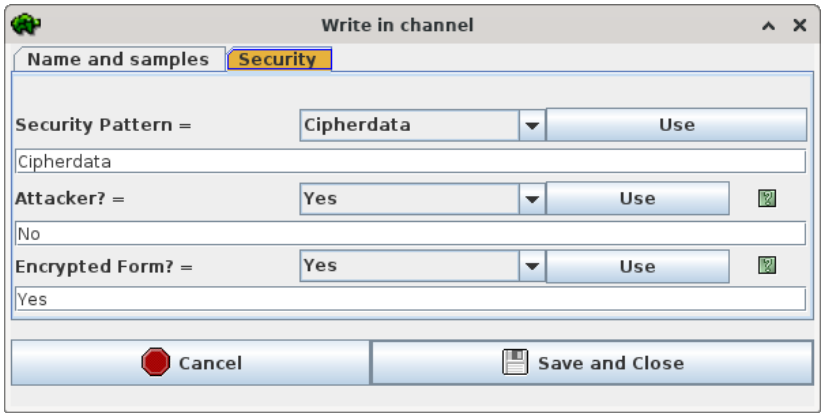


Figure 15: Channel security options dialog window

Let's now consider a third mapping (named "NonSecureArchitectureWithSecureFV") which basically consists in mapping the secure tasks to the non secure architecture (i.e., the

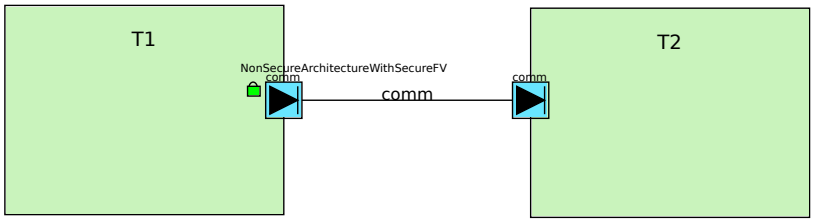


Figure 16: Verification result in the case of a non secure architecture but with secure functions

one with the public bus). The result of the security verification of this system is given in Figure 16. The confidentiality property is now verified.

Unfortunately, security mechanisms impact the performance of the system. TTool makes it possible to evaluate the performance of two different mappings,e.g. the one with no security (version 1), and the one with security mechanisms (version 3). To do this, TTool relies on the DIPLODOCUS simulator[5].

Without taking into account penalties of the hardware platform (e.g. cache miss, task switching time, boot time, etc.), simulation takes 20 cycles in the case of the non secure system (see Figure 17), and 220 (20 cycles + 100 cycles for ciphering + 100 cycles for deciphering) in the case of the secure mechanisms added to tasks(see Figure 18).

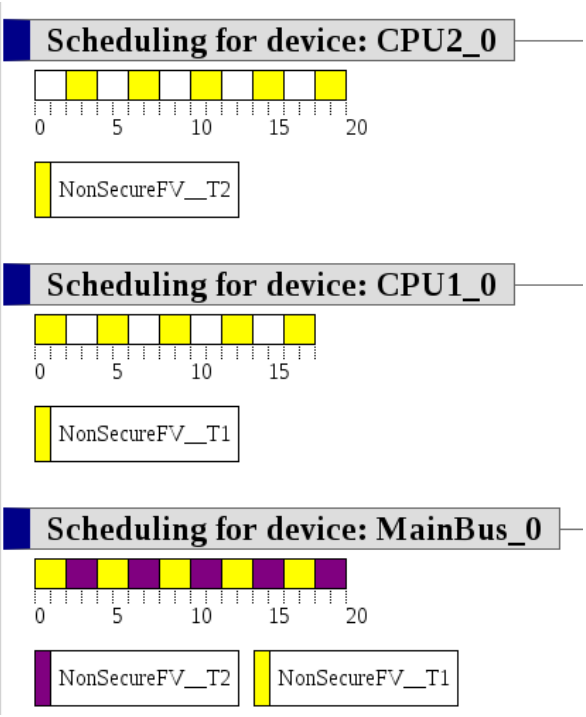


Figure 17: Simulation of non secure application mapped on the non secure architecture

[5]see the tutorial on DIPLODOCUS to learn how to use the DIPLODOCUS simulator: https://ttool.telecom-paristech.fr/diplodocus.html

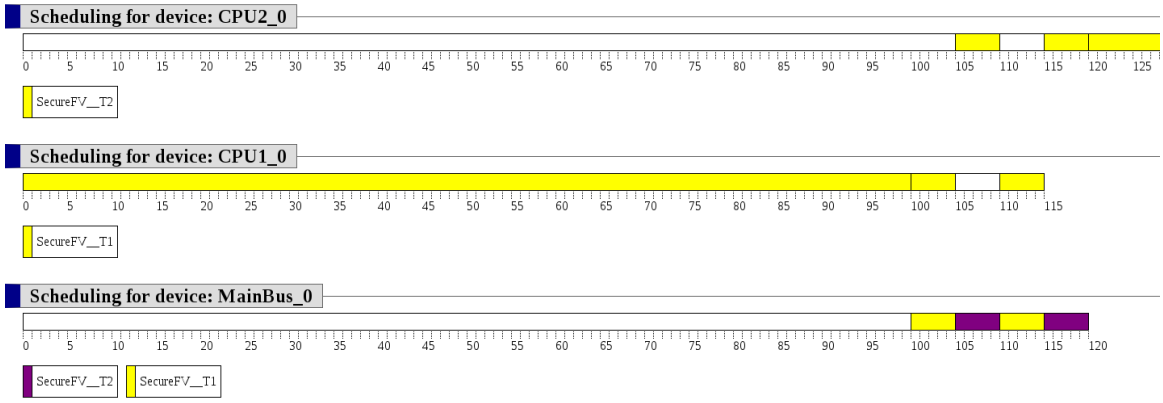


Figure 18: Excerpt of the simulation of the secure application mapped on the non secure architecture

## 3.8 Automatic Security Generation

Given security requirements and an unsecured model, our toolkit magically adds security elements. It can 1) add the security operators to a functional view, optionally with an added HSM performing all security operations, and 2) Automatically map keys securely. See the thesis of Letitia Li, 'Safe and Secure Model-Driven Design for Embedded Systems' for the detailed algorithms on how to add security.

First, each security-critical channel should be marked with whether the data across it should be checked for Confidentiality or Authenticity. It is assumed that the security properties to be checked are the ones that should be ensured for each channel.

### 3.8.1 Adding Security Operators

When the option to add security operators is selected, there are multiple options of the type of security operators to be added. The user should select if confidentiality, weak authenticity, and/or strong authenticity should be ensured for the model. For example, if the user only wishes to add operators to ensure confidentiality, then the toolkit will ignore the requirements on authenticity and only add the encryption operators to channels marked with the security annotation indicating that the data on them must be confidential.

In addition, for the operators being added, estimated times to perform encryption, decryption, calculate a MAC, etc, and the overhead, can be manually set in lieu of using the default options.

For example, using the insecure architecture and functional model with authenticity check from Figure 20 and 5, if the communication channel is marked that it should be authentic, and if we choose to ensure weak authenticity only, then the toolkit generates the functional model shown in Figure 21. The Message Authentication Code concatenated onto the message can only be calculated with the given key and message, and if T2 detects that they do not match, then it discards the message as it was not an authentic message sent by T1.

If instead we chose to add weak and strong authenticity, then the tasks should exchange a nonce to avoid replay attacks, and the functional model in Figure 22 is generated.

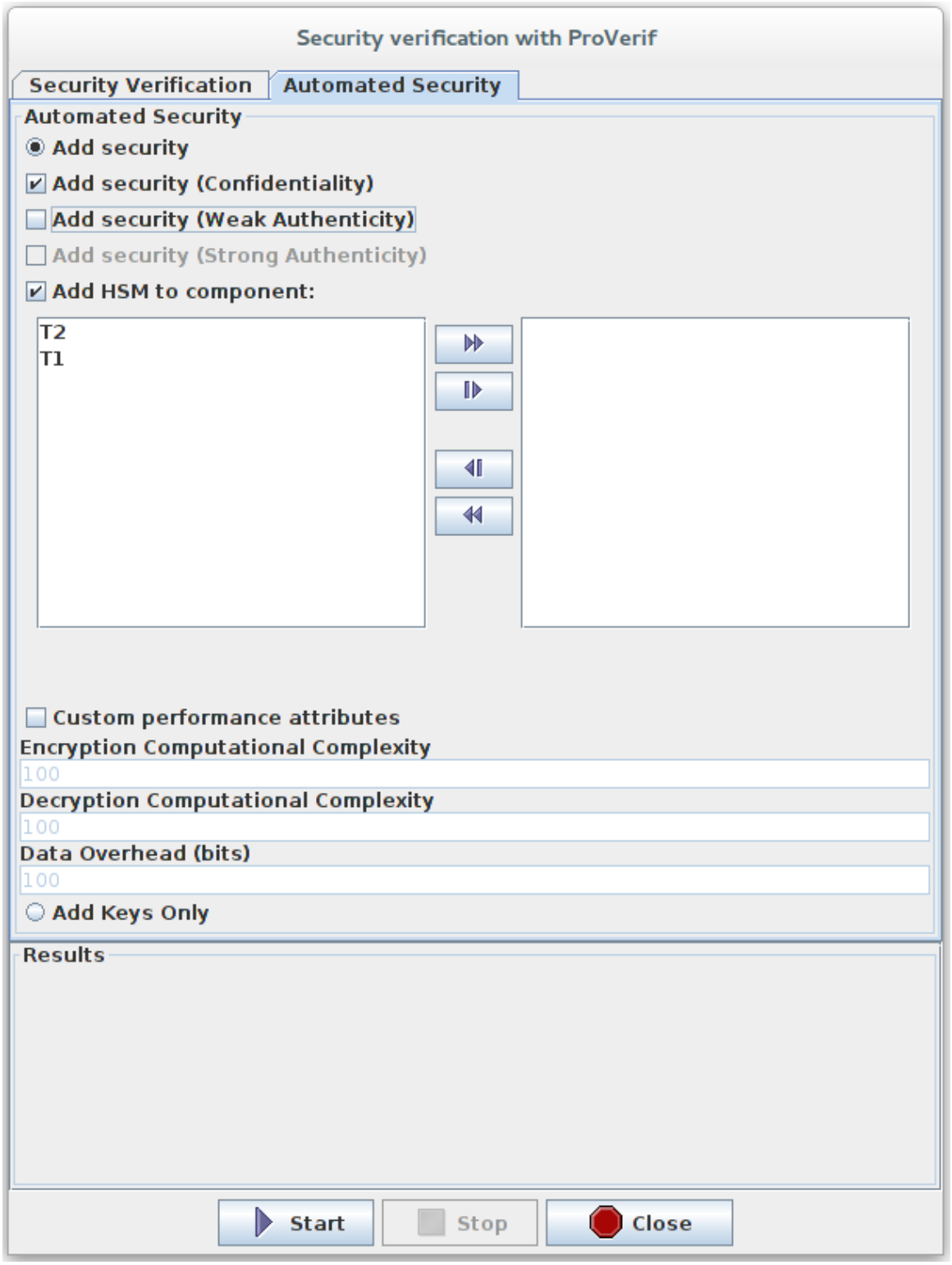


Figure 19: Window for Automatic Generation of Security

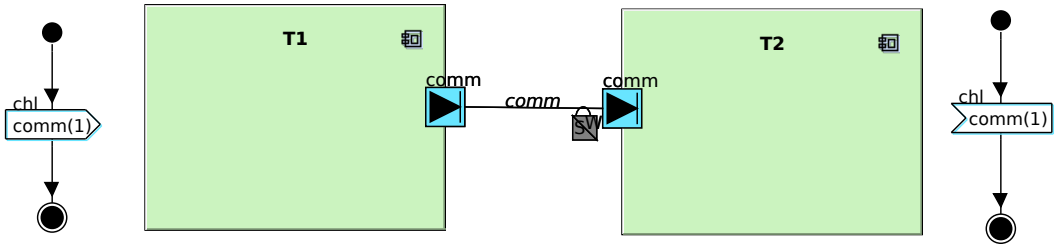


Figure 20: Functional View with authenticity check

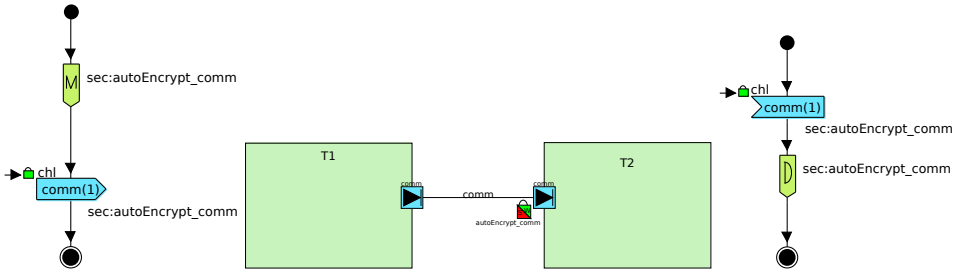


Figure 21: Functional view with automatically generated security operators to ensure weak authenticity

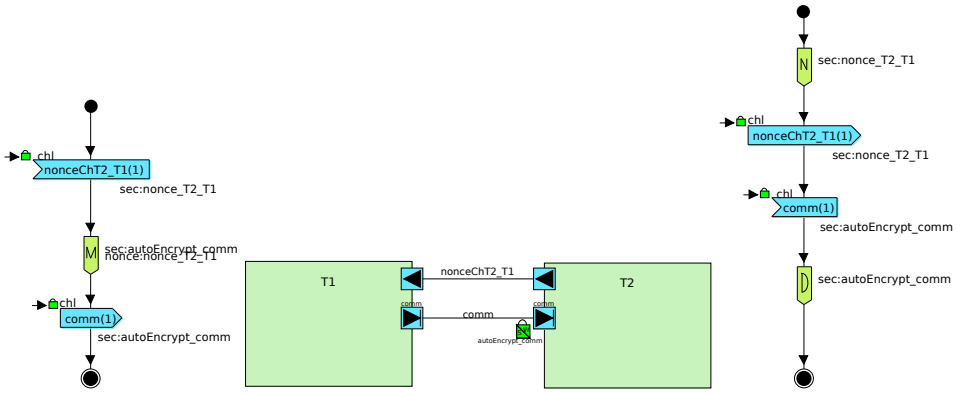


Figure 22: Functional view with automatically generated security operators to ensure strong authenticity

### 3.8.2 Adding Hardware Security Modules

For automatic security generation, there is also an option to add HSMs to perform all the security operators intead. Our toolkit can automatically add HSMs to designated tasks, including making all of the modifications to the diagrams relating to sending the data to the HSM, generating the HSM's activity diagram, etc. A single Hardware Security Module is added to each processor which executes at least one of the designated tasks. If multiple tasks mapped to a single CPU are designated to have a HSM added to them, then only a single HSM will be added.

For each HSM to be added to perform security operations for one or more tasks, first, the architectural diagram is modified to add a Hardware Accelerator and memory, with a connecting private bus.

Next, each task is modified, so that before each instance of sending a message which should be secure, the task first issues a request with the index of the channel (in the case of multiple channels to secure), and sends the data to the HSM. The HSM then performs the security operations, and returns the secured message to the task, which then sends the secured message to the receiving task. When a task receives data to be decrypted, it similarly sends the messages to the HSM, which then decrypts it and sends the message back, and which point the receiving task can understand the contents of the message.

For example, using the model in our example, chose to ensure confidentiality, and add a HSM to each task. A new HSM task is added for each HSM in the Functional model as shown in Figure 23, and a secure bus, memory, and Hardware Accelerator are added for each CPU on the Architecture/Mapping model as shown in Figure 24. Figure 25 shows how the activity diagram of task t1 is modified to send communications to the HSM to be encrypted the activity diagram of the HSM.

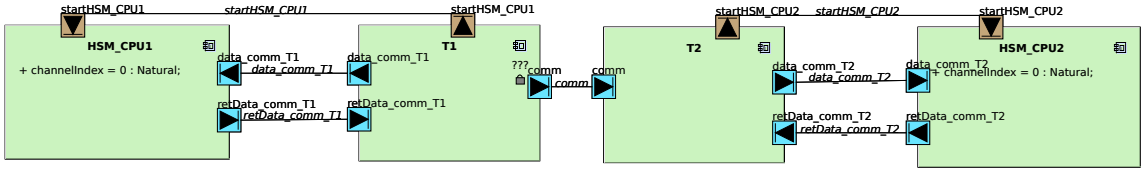


Figure 23: Functional Model with added Hardware Security Module Tasks

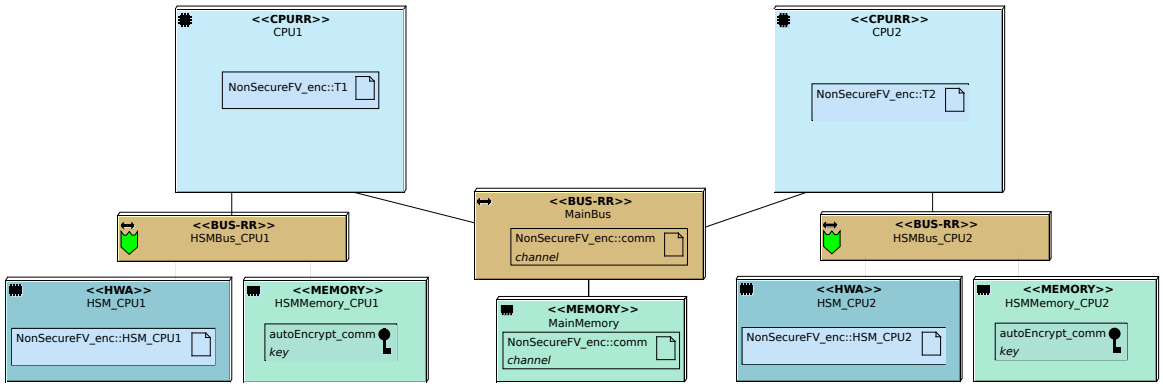


Figure 24: Architecture Model with added Hardware Security Modules

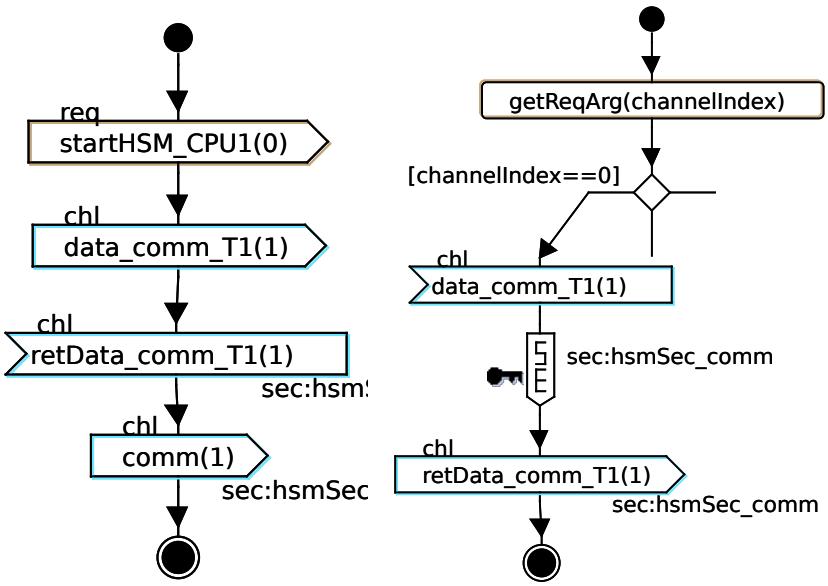


Figure 25: Modified Activity Diagram of T1 and HSM

### 3.8.3 Mapping Keys

With multiple Cryptographic Configurations, it may become tedious to map all of the keys to memory. Our toolkit therefore can find every Cryptographic Configuration used by a task, and then, depending on the type of the Cryptographic Configuration, map each applicable key to a memory that the task can securely access. For Cryptographic Configurations of type symmetric encryption or MAC, both the sending and receiving task will need to be able to access the key. For asymmetric encryption, however, all the sending tasks will need the public key while only the receiving task will need to access the private key.

At every security generation, keys are automatically generated and mapped securely. There is also the option to add keys alone after completing a design by hand.

If a key is sent along a bus accessible to an attacker, then the key would be known to the attacker, so we wish to avoid sending keys along public buses. For each task which needs the key, the algorithm searches for securely accessible memories from the processor to which it is mapped. The algorithm traverses all possible private buses and bridges using breadth-first search, until it finds a memory. The key is then mapped to that memory. If all possible secure paths are searched and no memories are found, then a warning is issued saying it is impossible to map keys for that task.

For example, for the automatically secured model in Figure 21, the keys for autoEncrypt_comm are mapped as shown in Figure 26, where there is a secure path to the memory. If they keys were mapped to MainMemory, then the attacker could recover the key when it was read over the public bus MainBus.

## 3.9 Designing security protocols

During the HW/SW partitioning stage, security mechanisms have been modeled at a high level of abstraction, mostly to place them correctly in the system, and to evaluate their impact on the system performance. During the software design stage, security protocols can be designed in a more precise way.

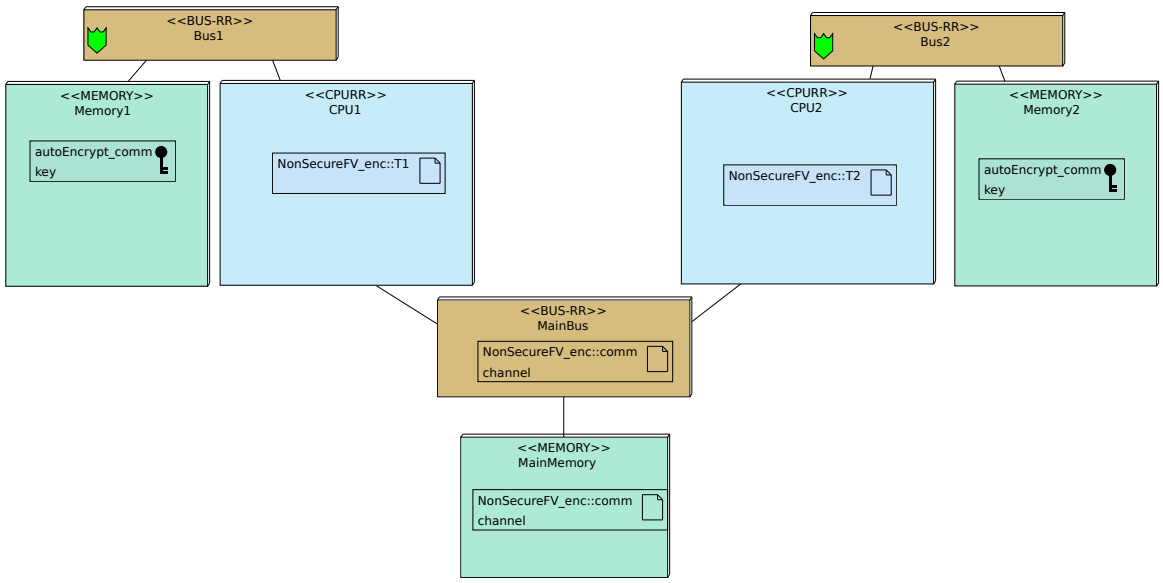


Figure 26: Mapping model with mapped keys

A software design contains a block diagram (see Figure 27) as well as a state machine for each task block (see Figure 28).

The block diagram contains a main block ("System") with two sub blocks ("T1" and "T2"). These tasks correspond to the same tasks modeled in the HW/SW Partitioning phase at a higher level of abstraction. Two other blocks "Key" and "Messages" are used to define custom data types. T1 and T2 are cryptoblocks, i.e. they define default cryptographic methods e.g. encrypt, decrypt, hash, mac, message manipulation (concat, cut), etc. Last, a pragma:

- links cryptographic keys of T1 and T2. This key "sk" is system-wide, which means that it is shared once for all protocol sessions.
- gives a security property to check: the value of the attribute "secretData" of "T1" must remain confidential.

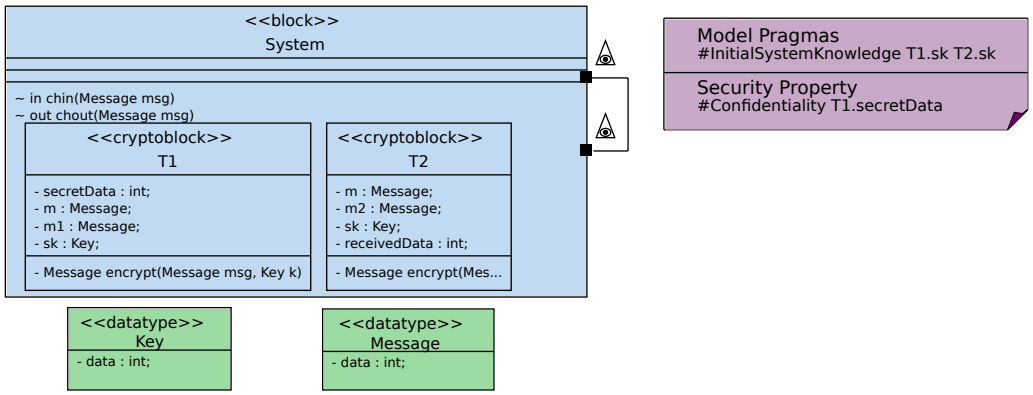


Figure 27: Design of a security protocol

The security formal verification can be performed from these diagrams. Just like for HW/SW partitioning models, both security properties and the reachability of states can be

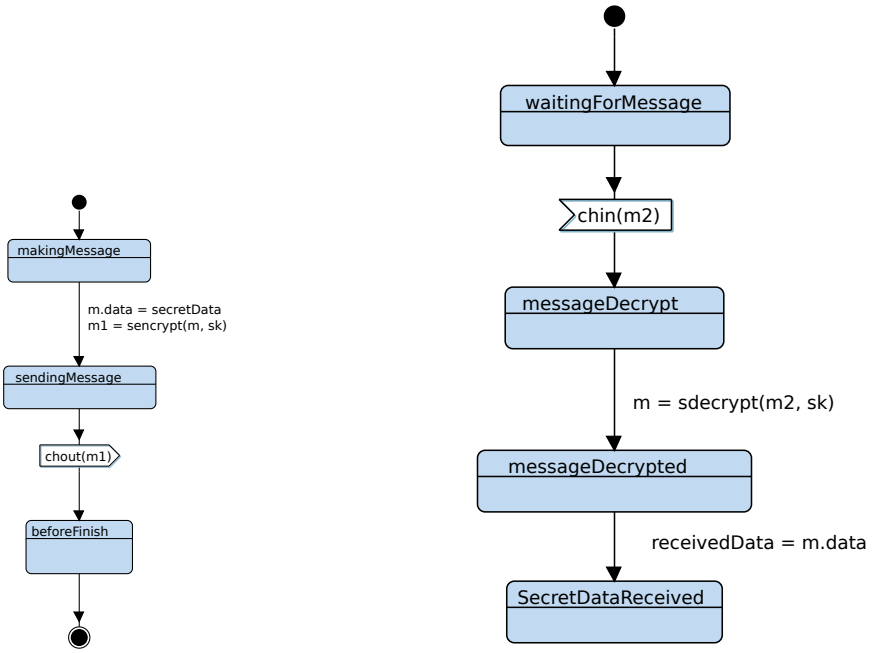


Figure 28: State machines of T1 (on the left) and T2 (on the right)

studied, and results are back-traced to the diagrams with e.g. green locks when the property is satisfied (see Figure 29).

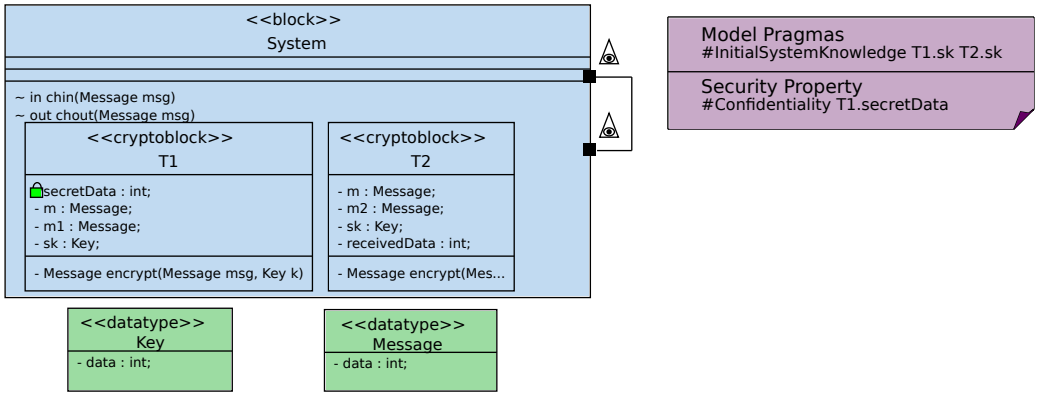


Figure 29: Result of the security verification