Radboud University Nijmegen

Bachelor Thesis

Model Checking Tamper-Evident Pairing

Author:
Manu Drijvers
(3040429)

Supervisors:
Marko van Eekelen
Rody Kersten

June 25, 2012

Abstract

Easily pairing wireless devices in a secure way has many real-life applications, such as in wireless sensor networks. The Wi-Fi Alliance designed the Wi-Fi Protected Setup to provide standard ways to easily pair wireless devices. Push button configuration is part of this standard and allows two devices to be paired by simply pressing a button on each device. However, this method is vulnerable to man-in-the-middle attacks. Tamper-evident pairing is a wireless pairing protocol that aims to solve the man-in-the-middle vulnerability of push button configuration, but only an informal proof of security is provided. In this thesis model checking in UPPAAL is used to analyze the security of tamper-evident pairing. We conclude that our model is secure, but future work is required to gain more certainty about the security of tamper-evident pairing.

Contents

1 Introduction **5**
- 1.1 Methods . . . 5
- 1.2 Motivation . . . 6
  - 1.2.1 Go-Green project . . . 6

2 Related work **7**

3 Theoretical framework **9**
- 3.1 Diffie-Hellman Key Exchange . . . 9
- 3.2 Attacker model . . . 9
- 3.3 Push button configuration . . . 10
- 3.4 Tamper-evident pairing . . . 10
  - 3.4.1 Out-of-band channel . . . 10
  - 3.4.2 Tamper-evident announcement . . . 11
  - 3.4.3 Bit-balancing algorithm . . . 12
  - 3.4.4 Using TEAs . . . 12

4 Uppaal **13**
- 4.1 Timed automata . . . 13
- 4.2 Networks of timed automata . . . 13
- 4.3 Uppaal components . . . 13
- 4.4 Coffee machine example . . . 14

5 Modeling **17**
- 5.1 TEA model . . . 17
  - 5.1.1 Sender . . . 17
  - 5.1.2 WirelessMedium . . . 17
  - 5.1.3 Adversary . . . 17
  - 5.1.4 Receiver . . . 18
- 5.2 TEP model . . . 19
  - 5.2.1 User . . . 19
  - 5.2.2 Enrollee . . . 19
  - 5.2.3 Registrar . . . 19
  - 5.2.4 Adversary . . . 20

6 Model checking **21**
- 6.1 TEA model checking results . . . 21
  - 6.1.1 Model checking parameters . . . 21
  - 6.1.2 Results without adversary . . . 21
  - 6.1.3 Results with adversary . . . 22
- 6.2 TEP model checking results . . . 22
- 6.3 Evaluation of results . . . 23

7 Conclusion **25**

8 References **27**

Appendix A TEA model **29**

Appendix B TEP model **42**

1 Introduction

More and more devices are capable of communicating over wireless networks. To securely connect these devices, pairing protocols are developed. Not all of these devices have a display or keyboard, so the Wi-Fi Alliance designed push button configuration (PBC) as part of their Wi-Fi Protected Setup standard. Using PBC, one can pair two devices by pressing a button on each device. Unfortunately, this method is not secure, as it is vulnerable to man-in-the-middle attacks. Tamper-evident pairing (TEP) is based on this method, and introduces a new primitive, the tamper-evident announcement (TEA). TEP claims to have solved this issue, but only provides an informal proof. This raises the research question: Is tamper-evident pairing vulnerable to man-in-the-middle attacks?

To answer this question we have to answer the following subquestions:

1. (a) How can a tamper-evident announcement be modeled?
   (b) How can an attacker be integrated in the model of a tamper-evident announcement?
2. (a) How can tamper-evident pairing be modeled?
   (b) How can an attacker be integrated in the model of tamper-evident pairing?
3. How can the results of model checking be interpreted?

1.1 Methods

There are multiple approaches to analyze security protocols. The two general ways are model checking and theorem proving. In model checking, a property is checked for a finite number of states, where in theorem proving, all possible states are considered. UPPAAL [3] is an example of a tool for model checking. Prototype Verification System [15] is an example of a theorem prover. A special form of theorem proving is type checking. The spi calculus is the pi calculus extended with types for security properties [1]. ProVerif [12] is a type checker that can type check spi calculus, to check security properties. In some theorem provers, like ProVerif, protocols are considered on a high abstraction level. Tamper-evident pairing must be checked on a lower abstraction level because it dictates implementation decisions. Other theorem provers might be suitable, but constructing such a proof requires a lot of effort and time. Model checking is less time consuming and more fit for the scope of this thesis. UPPAAL is designed to model timed systems, and timing is a very important aspect of TEP. It has also been shown that UPPAAL can be used to analyze security protocols [4]. Therefore UPPAAL will be used to analyze TEP.

First the tamper-evident announcement will be modeled in UPPAAL. This models the transmission of a single TEA. The desired properties of a TEA are translated to UPPAAL queries, and these queries are tested on the model. If some query does not hold, TEP is broken. If all desired properties of a TEA hold, TEP will be modeled. This will be a separate model using just the results of the TEA modeling, to make sure that this model will be computationally tractable. The model will contain a state in which a succesful man-in-the-middle attack

has been performed. If this state is reachable, an attack will be found by tracing the steps it took to get there. Otherwise, it is likely that there is no attack, but a formal proof is required to be sure.

1.2 Motivation

Model checking cannot provide a formal proof of security, but it may find vulnerabilities if they exist. If no vulnerabilities will be found, then attempting to formally prove the security will be the next step. The intention is that this proof will also be constructed within the Go-Green project (1.2.1). The results of this thesis will therefore support a formal proof.

Proving that TEP is secure would be useful for many real life scenarios. TEP could replace PBC as standard push button wireless pairing mechanism to provide better security. A wireless pairing mechanism is needed for medical devices [16][9], where security is of vital importance. Another application of a secure pairing mechanism is the Go-Green project.

1.2.1 Go-Green project

The Go-Green project is a collaboration among a number of companies and universities, including Delft University of Technology, University of Twente and Radboud University. The goal of this project is to make

> "[...] an intelligent energy-aware system that learns people's behavior, understands their needs, provides reliable feedback in terms of best practices for energy use and distribution, intelligently harvests energy from various sources, and allows people to be in control." [18]

To achieve this goal, the project aims to develop a system that gathers data with sensors throughout the house. This data is processed by a smart system and combined with external data such as the weather forecast. The system will help the user control the energy consumption based on this information and user feedback.

This system depends on sensors that can send their data securely, as privacy sensitive data may be sent. To connect all these sensors, a simple and secure way to connect devices is required. If TEP is proven secure, then this may be used as pairing mechanism. I will only consider the security of TEP, the other privacy and security aspects of this project are not in my scope.

2 Related work

Wi-Fi protected setup has been analyzed a lot. Viehböck found an attack on the PIN authentication [20]. The PIN consist of eight digits, but the last one is a checksum digit. A brute-force attack would require at most 10^7 attempts. Viehböck discovered that from the registrar's response to a connection attempt can be derived whether there was a mistake in the first or second half of the PIN. This enables an attacker to first brute-force the first half, and then the second half. The first half has 10^4 options, the second half 10^3. Using this information, a brute-force attack only requires $10^4 + 10^3 = 11000$ attempts instead of $10^7 = 10000000$ and is feasible.

Corin et al. [4] demonstrated how Uppaal can be used to verify security protocols in a real time setting. This is fundamentally different than the tranditional security in which time is not considered. A Dolov-Yao [7] style intruder is modeled, but it became more powerful than a Dolov-Yao attacker by modeling it as a timed automaton, because it is now aware of time. Time also introduces new possibilities for security protocols, because sending a message at a specific moment instead of some other moment carries information. Corin et al. described this as a preliminary idea, but the on-off slots in a TEA are using a similar principle.

Kuo et al. [11] analyzed bluetooth simple pairing and the WPS protocols. They conclude that an attack on PBC is very likely to succeed, since they do not make use of an out-of-band channel and use unauthenticated Diffie-Hellman.

3 Theoretical framework

Firstly the required theoretical concepts will be introduced. We start by describing Diffie-Hellman key exchange (3.1) as this is used in PBC and TEP. Then the attacker model (3.2) is provided, followed by descriptions of PBC (3.3) and TEP (3.4).

3.1 Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange [5] allows two users to establish a shared key. Both parties know a non-secret prime p, and a non-secret generator of multiplicative group of integers modulo p named g. The protocol works as follows:

$$A \rightarrow B : g^{S_A} \ mod\, p$$

$$B \rightarrow A : g^{S_B} \ mod\, p$$

Where S_i is a secret of user i. Now the shared key K_{AB} is $g^{S_A S_B} \ mod\, p$. A and B can compute this key by raising the received message to the power of their secret. An eavesdropper must be able to compute the discrete logarithm to acquire the key. However, we do not know an efficient way to compute the discrete logarithm, and assume this to be hard [13]. Therefore we assume that an eavesdropper cannot efficiently compute the key.

Diffie-Hellman Key Exchange is vulnerable to man-in-the-midle (MITM) attacks on a wireless medium, because an adversary can hide a message and send another message instead. An attack would look like this:

$$A \rightarrow E : g^{S_A} \ mod\, p$$

$$E \rightarrow B : g^{S_E} \ mod\, p$$

$$B \rightarrow E : g^{S_B} \ mod\, p$$

$$E \rightarrow A : g^{S_E} \ mod\, p$$

Now E shares a key with A and with B. There are authenticated protocols based on Diffie-Hellman key exchange that do not have this vulnerability, like station-to-station protocol [6]. However, these protocols require shared keys for authentication. In push button configuration we are trying to establish a shared key, so authenticated protocols cannot be used.

3.2 Attacker model

The security of tamper-evident pairing will be analyzed using the attacker model used by Gollakota et al. [8]. A Dolev-Yao [7] like attacker is used, which is an attacker that has full control over the medium. This means an attacker can eavesdrop any message from the medium, transmit with arbitrary power and thus overpower any message and replace it with his own message. The attacker used differs from a Dolev-Yao as he cannot remove messages sent by others. This means the attacker cannot make a busy wireless medium look like an idle wireless medium.

We assume perfect hash functions, so an attacker cannot efficiently find collisions or determine the preimage of a hash.

3.3 Push button configuration

To be able to securely add wireless devices to a Wi-Fi network, the Wi-Fi Alliance introduced the Wi-Fi Protected Setup (WPS) standard [19]. Push button configuration (PBC) is a part of WPS and lets two devices pair when a user presses a button on both devices. This is especially useful for small devices that do not carry the interface to enter a password.

A new device called enrollee wants to enroll on the network. The other device, called the registrar, can give credentials to join the network. The user presses the button of both devices within the so-called walk time, which is two minutes. When the enrollee's button is pressed, he will actively search for a registrar for the duration of the walk time. If it finds more than one registrar an error will occur. Pressing the button of the registrar causes it to first check how many enrollees tried to connect in the last two minutes (monitor time), if more than one enrollee tried to connect, an error is given immediately. If not, it will listen to enrollees trying to connect for the duration of the walk time. If only one enrollee tried to connect at the end of the walk time, the registrar will answer according to the registration protocol. A shared key is established using Diffie-Hellman Key Exchange, and the credentials for the network are sent to the enrollee. If multiple enrollees tried to connect, an error will be given.

An attacker can perform a man-in-the-middle attack by jamming the enrollee's connection attempt, transmitting his own connection attempt simultaneously or sending a connection attempt and continue hogging the medium afterwards, making it impossible for an enrollee to send his connection request. The registrar will only notice one connection attempt, and will pair with the attacker. Then the attacker can impersonate a registrar to the enrollee, so the attack will not be detected.

3.4 Tamper-evident pairing

Gollakota et al. (2011) modified PBC to prevent MITM attacks. This new protocol, named tamper-evident pairing or TEP, hopes to provide simple and secure wireless pairing without the use of out-of-band channels. To achieve this, a new security primitive has been designed, named a tamper-evident announcement (TEA).

3.4.1 Out-of-band channel

When an out-of-band channel is used in authentication, a trusted communication channel other than the channel requiring authentication is used. In internet banking text messages are often used to send a confirmation code as extra layer of security. The cellular network is the out-of-band channel in this example. Other examples of out-of-band channels are audio channels, visual channels, near field communication, or a USB flash drive that is physically transferred between devices. Using out-of-band channels can provide more secure authentication [10], but also increases device manufacturing costs, since another channel has to be supported.

3.4.2 Tamper-evident announcement

The TEA is a new security primitive. The goal of a TEA message is that an attacker cannot hide the message, nor can he modify its content. To achieve this, the fact whether there is energy on the channel at certain moments is used. The key is that an adversary cannot remove energy from the medium.

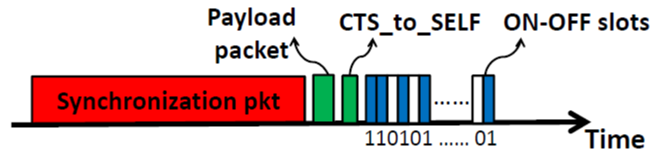


Figure 1: The format of a TEA [8]

A TEA starts with an exceptionally long packet called the synchronization packet, followed by the payload packet. After the payload a CTS_TO_SELF is sent. This is a message defined in the 802.11 specification that informs the nodes that the sender of the CTS_TO_SELF is about to transmit data, and tells others to refrain from transmitting. By sending this message, all honest nodes will remain silent for the duration of the TEA. Then a sequence of equally sized packets called slots is sent. A slot can either be the transmission of random data, or completely silent. When you interpret a transmission slot as a 1 bit and a silent slot as a 0 bit, the slots together must form a hash of the payload.

The synchronization packet attempts to prevent that an attacker can hide the message by creating a collision, because such a long collision is easily detectable. This packet also indicates the start of a TEA. This is required since timing is important when receiving the slots. The hash slots are intended to prevent an attacker from modifying a message by transmitting simultaneously but with a higher power. Since an attacker can not remove messages from the medium, he can not turn a transmission slot into a silent slot. An attacker can still change the hash by turning a silent slot into a transmission slot by transmitting data in this time interval. To counter this, Gollakota et al presented a bit-balancing algorithm that transforms data to have an equal number of zeros and ones. By using a bit-balanced hash, the attacker can no longer change a silent slot into a transmission slot, because this destroys the balance and makes the hash invalid. The CTS_TO_SELF makes sure that no honest node will change a silent slot into a transmission slot.

In the implementation of TEP proposed in [8], to detect whether a slot was a transmission slot or a silent slot, the registers AR5K_PROFCNT_CYCLE and AR5K_PROFCNT_RXCLR from the ath5k firmware are used. Every clock cycle, AR5K_PROFCNT_CYCLE is incremented. AR5K_PROFCNT_RXCLR is incremented on a clock cycle if the hardware finds energy during that clock cycle. A sensing window is an interval in which these two registers are analyzed. At the start of every sensing window AR5K_PROFCNT_CYCLE and AR5K_PROFCNT_RXCLR get reset. At the end of a sensing window, $\frac{AR5K_PROFCNT_RXCLR}{AR5K_PROFCNT_CYCLE}$ is stored. This fraction, called the fractional occupancy, indicates the occupancy of the wireless medium during the sensing window. Perfect synchronization between sender and receiver is hard to achieve. If the receiver would use 40 μs sensing windows, a part of the wrong slot may

be measured in a sensing window due to non perfect synchronization. To solve this problem, 20 μs sensing windows are used for the 40 μs slots. By using 20 μs sensing windows, one of the two sensing windows for a slot will be entirely in that slot. At the end of a TEA, the receiver has two fractional occupancy values for each slot. Either the even or the odd sensing windows are the ones entirely in a slot. To determine which ones are, the variance of the fractional occupancy values is calculated for the even and the odd windows. The sensing windows that are entirely in one slot have the highest variance, because overlapping decreases the variance. Finally the receiver has to determine whether a slot was a transmission slot or a silent slot given the fractional occupancy. This is done by simply comparing it to a threshold value. This threshold value is unspecified by Gallakota et al.

3.4.3 Bit-balancing algorithm

This algorithm takes a bit sequence of length N, where N is even, and returns a bit sequence of length $N + 2\lceil \log N \rceil$. It checks the difference between zeros and ones in the input. If this difference is not zero, the first bit is flipped. The difference is calculated again, and if unequal to zero, the next bit is flipped. At the ith iteration, the first i bits are flipped. This will be repeated until the difference is zero.

The difference will always become zero. If initialy the difference is d. d is even because N is even. Every time you flip a bit, d increases or decreases 2, which means that the difference will remain even. The difference after N iterations, i.e. all bits flipped, is $-d$. Zero is even and between d and $-d$, and will therefore be reached.

The algorithm returns the input with the first i bits flipped, concatenated with the binary representation of i in manchester encoding [17]. Manchester encoding turns a 0 into 10 and a 1 in 01, so the encoding of i is bit-balanced. The length of the encoding of i is $2\lceil \log N \rceil$, because the binary representation of i is $\lceil \log N \rceil$, and the manchester encoding doubles the length. Now the result has the same amount of zeros and ones, and its length is $N + 2\lceil \log N \rceil$.

3.4.4 Using TEAs

TEP works like PCB, but probe requests and responses are now TEAs containing the Diffie-Hellman public key of the sender. If an enrollee or registrar receives a tampered message it will continue sending requests or responses, but after the duration of the walk time it will not pair.

4 Uppaal

UPPAAL is a tool developed by Uppsala University and Aalborg University [3]. UPPAAL is used to verify real-time systems by modeling them as networks of timed automata.

4.1 Timed automata

A timed automaton is a finite automaton extended with a finite set of clock variables and invariants [2]. All clocks progress synchronously, but each clock can individually be reset. Transitions can use contraints on clock values, e.g. transition x can only be used when clock c is greater than five. Invariants can be added to states. An invariant is a boolean expression on clocks that must be invariantly true. This makes it possible to reason about systems in which timing is a crucial aspect.

4.2 Networks of timed automata

UPPAAL models are networks of timed automata. Such a network consists of multiple timed automata. Bounded discrete variables and synchronization channels are added to let the timed automata communicate. The variables are part of the state and can be used in the same way as they are used in programming languages, and one can write user-defined functions using variables. Transitions and invariants can now also put contraints on variable values or function results.

Synchronization channels are used to enforce transitions in multiple timed automata to be enabled at once. A synchronization channel must be declared first. If an edge is labelled with `c!`, where c is a synchronization channel, another edge labelled with `c?`, both edges are enabled at the same time. The default way of synchronization is binary synchronization. In binary synchronization, when there are multiple edges labelled `c?` one of those is chosen nondeterministically. In broadcast synchronization, a single `c!` enables all edges labelled `c?`.

For individual states it is possible to mark them as urgent. If any of the timed automata is in an urgent state, time may not progress. States can also be marked as committed. A committed state freezes time like an urgent state, but gives edges from this state priority over edges from non-committed states. If there is an automaton in a committed state, the next transition must involve an edge from one of the committed states.

4.3 Uppaal components

UPPAAL consists of three major components: the editor, the simulator and the verifier.

The editor is used to design a model. A model contains templates. Every template is a timed automaton with it's own variables and function declarations. An UPPAAL model also contains global declarations, the variables and user-defined functions are accessible from all templates. In the system declaration, one can define of which templates a system consists. For example, when modeling a railroad crossing, one might make a crossing template and a train template, and the system would consist of one instance of the crossing template and two instances of the train template.

The simulator lets one use the model. The user can choose which transitions take place and analyze the values of the variables in the state. The simulator is useful to explore the model and check whether it functions as intended. Another feature of the simulator is stepping through a trace. A trace is a chain of transitions that the verifier may provide. The simulator can be used to analyze this trace.

The verifier checks whether a given query holds in the model. Queries use temporal logic [14]. Queries often look like `A[] expression`. The `A` means the expression must hold for all possible paths in the model. The `[]` means that the expression must hold for every single step in the path. Therefore the combination `A[] expression` means that expression always holds in the model. A query may also start with `A<>`. The `A` still means that the expression must hold in all paths, but `<>` means that expression must eventually hold in a path, so the combination `A<>` means that for every path the expression must eventually be true. Instead of verifying an expression for all paths, one can whether there exists a path in which a property holds. This can be done using an `E` instead of an `A`. `E[]` expression means that expression must hold in all states of some path, and `E<>` expression means that expression must eventually hold in some path. The last form or queries is `p --> q`, which means that whenever p holds, q must eventually hold.

When the verifier checks an `E[]` or `E<>` query that holds, it will provide the witnessing trace, and when a `A[]`, `-->` or `A<>` query is check that does not hold, a counter-example trace will be given.

4.4 Coffee machine example

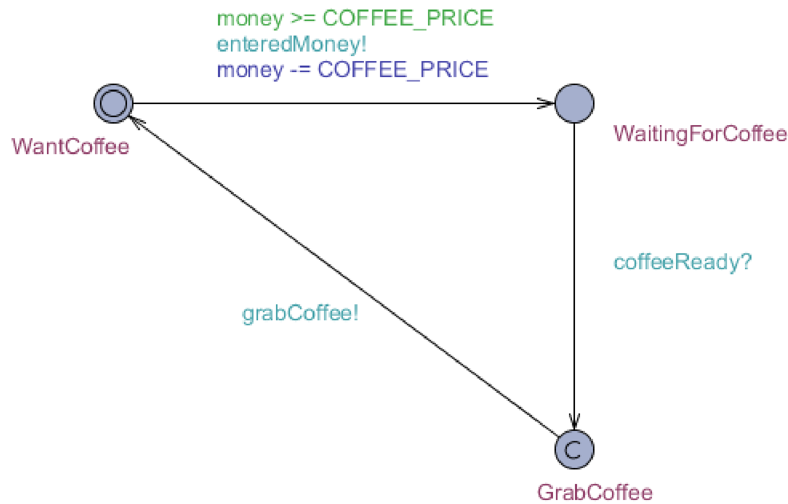


Figure 2: The User template of the coffee machine example

To make the UPPAAL concepts more clear a coffee maker example model is provided. The model consists of two templates, called User (figure 2) and CoffeeMachine (figure 3), and models a user that enters money in a coffee machine,

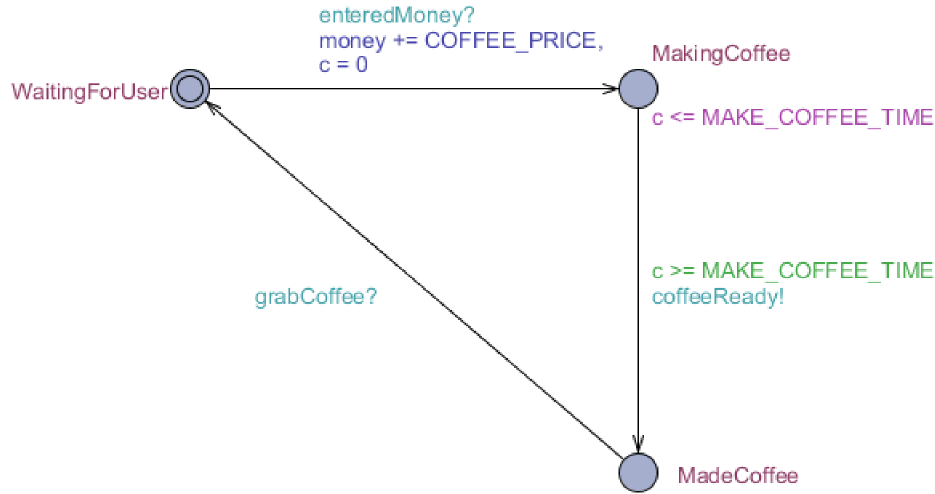


Figure 3: The CoffeeMachine template of the coffee machine example

waits for his coffee and gets his cup of coffee. After buying a cup of coffee, he might want another cup of coffee and start over.

The user starts in the state `WantCoffee`, which is indicated by the second circle in this state. This state has one outgoing edge, which models the entering of money in the coffee machine. The transition has a guard field with `money >= COFFE_PRICE`, preventing this edge from being taken when the user doesn't have enough money. The update field of this transition states `money -= COFFEE_PRICE`, which lowers the user's money with the coffee price, modeling the fact that the user entered the money in the machine. The transition is synchronized with `enteredMoney!`. The CoffeeMachine template starts in state `WaitingForUser`, and has an outgoing edge with `enteredMoney?` in the synchronization field. Now these two transitions can be taken simultaneously. The CoffeeMachine transition has `money += COFFEE_PRICE, c = 0` which increments the machine's money, and resets a clock variable `c`. The user is now in state `WaitingForCoffee`, with an outgoing edge that synchronizes on `coffeeReady?`, modeling the fact that he is waiting for the coffee to be ready. This transition cannot be taken without the machine being ready, as this binary synchronization channel needs a `coffeeReady!` to be enabled. The CoffeeMachine is now in state `MakingCoffee`. From this state there is one outgoing edge to `MadeCoffee`, but this is guarded with `c >= MAKE_COFFEE_TIME` which prevents the transition being taken before the time it takes to make coffee has passed. The state `MakingCoffee` has an invariant `c <= MAKE_COFFEE_TIME` that prevents CoffeeMachine from staying in this state any longer than the time it takes to make coffee. Without this invariant it could remain in this state for as long as he wants, and a coffee machine that takes an arbitrary amount of time to make coffee does not make sense. The invariant combined with the guard on the transition force the coffee maker to take the transition at exacty `MAKE_COFFEE_TIME`. Now we have both `coffeeReady!` and `coffeeReady?` so

both transitions are taken simultaneously. The user is now in state `GrabCoffee` and the machine in state `MadeCoffee`. Because `GrabCoffee` is a committed state, time may not progress and the transitions from this state have priority. This models the fact that a user would take his coffee from the machine as soon as it is ready. The transition synchronizes on `grabCoffee!`. The coffee machine has a transition with `grabCoffee?` allowing the two transition to be taken together. Both templates are back in the intial state. The coffee machine is ready for another user, and the user might buy another cup of coffee.

A system is composed using two users and a single coffee machine. The system declarations look like:

```
User1 = User();
User2 = User();
CoffeeMachine1 = CofffeeMachine();
system User1, User2, CoffeeMachine1;
```

One might want to verify the fact that the total amount of money in the system remains the same, i.e. no money appears or disappears. This can be checked using the query

```
A[] User1.money+User2.money+CoffeeMachine1.money ==
        2*USER_INITIAL_MONEY + MACHINE_INITIAL_MONEY
```

where `USER_INITIAL_MONEY` is the amount of money each user initially has, and `MACHINE_INITIAL_MONEY` is the amount of money initially in the machine. This query checks whether for all reachable states the total amount of money is equal to the initial amount of money in the system. The UPPAAL verifier responds with `property is satisfied`, indicating that this property holds.

5 Modeling

Two models have been designed, one for TEA and one for TEP. In the TEP model we assume that TEAs work as intended. This split has been made to decrease the model checking complexity. The TEA model is described in section 5.1, the TEP model in section 5.2.

5.1 TEA model

The TEA model consists of four templates, each of which is described in it's own section.

5.1.1 Sender

One of the templates of the TEA model is the sender. To send a TEA, the sender must know a payload, must be able to calculate the hash of this payload and be able to bit-balance this hash. The payload could be chosen nondeterministically, but this increases the possibilities and model checking time a lot. Therefore the payload must be given. Calculating the hash and the bit-balanced hash are tasks that the sender and receiver do by themselves, and timing and communication do not matter. In UPPAAL, this can be modeled as a user-defined function. Because we assume a perfect hash function and we are not looking for an attack using colissions is implementing a hash function not required. The bit-balancing is required, and is therefore implemented, see appendix A.1.

The sender starts by determining the payload, the hash and the balanced hash. Then the syncpacket is sent, this is modeled using a broadcast channel. After the synchronization packet the sender sends his payload. Then the SendingHash state is reached in which the balanced hash will be sent using the slots. A counter keeps track of how many slots have been sent. The invariant on the state SendingHash combined with the clock guards on the transitions to TransmissionSlot and SilentSlot force the sender to send or remain silent at fixed times.

5.1.2 WirelessMedium

A sender or adversary can send data by using the synchronization channel send. The data is passed using a global variable called `dataToSend`. The sender stores his payload on this variable and calls `send!`. The medium will reset his clock. Firstly the adversary may edit the data, as this is part of the attacker model. The possibly edited data is now stored on the global variable `dataInAir`, and the broadcast channel `transmit` is used. Now everyone listening to the transmit channel know that data is sent, and this data can be retrieved from the variable `dataToSend`. Boolean variable `mediumBusy` is set to true, indicating that the wireless medium is currently transmitting. It will remain in state busy for the time it takes to send the data. After that it returns to the idle state and mediumBusy is set to false.

5.1.3 Adversary

As described in the attacker model section, an adversary can change the content of any message. This is modeled using a synchronization channel named

`captureEffect`. When the wireless medium receives a message to send, it lets the adversary change the message contents. According to the attacker model, the content can be changed to any value. This would dramatically increase the number of possible transitions and the model checking complexity. Therefore the adversary is limited to changing the contents to a predefined value or not changing it at all. If the adversary is able to replace message contents with any value then model checking would find attacks using a hash collision. Since a perfect hash function is assumed, these attacks are not considered. Therefore limiting the adversary to a predefined value does not prevent attacks.

The adversary can also transmit data. By enabling the adversary to transmit for any given moment checking the model is not computationally tractable. To counter this problem, the adversary may only transmit when the medium is idle. Since the receiver only checks whether the medium is busy or idle and does not consider the data that are transmitted is there no difference between two nodes transmitting simultaneously and a single node transmitting. Using this limitation the adversary has still too many options for the model to be computationally tractable. To limit the options even further the adversary can choose between predefined time frames for every slot. For example, the possible time frames can be chosen as (0,0), (0,20), (20,40), (0, 40) where the first first element of a tuple denotes the time to start transmitting in a slot and the second element the time to stop transmitting, the adversary can respectivly choose to not transmit at all, transmit during the first half of a 40 μs slot, transmit during the second half of a 40 μs slot or transmit during the entire slot. By varying the number of allowed time frames the right number of possibilities can be found, i.e. the highest number for which the model is still computationally tractable.

The adversary has a boolean variable `tampered` that is set to true whenever he performed any tampering, i.e. editing payload and transmitting data during silent slots. This is not modeling any behaviour, it is just useful for model checking as will become clear in section 6.1.

5.1.4 Receiver

The receiver will wait for a synchronization packet, which is modeled as a broadcast channel. The sender starts by sending the payload, which is stored by the receiver. Then the slots are analyzed. It uses 20 μs sensing windows, in which it repeatedly checks whether the medium is busy or idle. The frequency of this polling can be configured. At the end of a sensing window, the fractional occupancy must be stored. An approximation is required because UPPAAL does not support float variables. To make an approximation, the numerator is multiplied by a constant factor c. Then a user-defined function that provides rounded integer division is used. So instead of storing the fractional occupancy, we store $round(c \cdot fractionaloccupancy)$. After the TEA, the variance of the even and odd fractional occupancies are calculated using a user-defined function and compared. The factor c does not have to be removed, because

$$Var(c \cdot X) = c^2 \cdot Var(X)$$

and thus for $c \geq 0$

$$Var(c \cdot Even) < Var(c \cdot Odd) \leftrightarrow Var(Even) < Var(Odd)$$

A boolean array is derived from the fraction occupancies with the highest variance by checking if $round(c \cdot fractionaloccupancy) \geq 0.5 \cdot c$, and this array is compared to the balanced hash of the received payload. If equal, the receiver will reach state `Valid`, indicating the received TEA is valid. If unequal, the receiver will go to state `Tampered`, indicating that the received TEA has been tampered with.

5.2 TEP model

The TEP model consists of four templates: a user pressing the buttons, an enrollee, a registrar and an adversary. Each template will be described individually in the following sections. In this section we assume that TEAs work as intended.

5.2.1 User

The user may start by pressing the enrollee or the registrar button. Within the walk time the button of the other device is pressed. The buttons are modeled as synchronization channels.

5.2.2 Enrollee

When the enrollee button is pressed TEA mode will be entered. In TEA mode the enrollee repeatedly transmits probe requests and listen to probe responses. These messages are modeled using shared data fields and broadcast channels. Requests and responses have a boolean field `tampered` indicating that this message has been tampered with. This is accurate since these messages are sent using TEAs. Whenever a probe response is detected, a user-defined function is called that handles responses. The registrar Diffie-Hellman key will be stored and it checks the validity of the message. If the `tampered` field of this response is true then a local variable `tampered` is set to true. After the duration of the walk time the enrollee will leave TEA mode. If the `tampered` variable is true, the enrollee will go to state `TAMPERED`. Otherwise, he will check the amount of registrars detected. If exactly one registrar is found, he will move to state `PAIR`, indicating that the enrollee is willing to start the WPS registration protocol using the Diffie-Hellman key of the single registrar he detected. If zero or more than one registrars were detected, the enrollee will move to states `NO_REGISTRAR` and `SESSION_OVERLAP` respectively.

5.2.3 Registrar

The registrar always listens to probe requests. Whenever a request is detected, the Diffie-Hellman key and the time of receiving this key are stored. It also checks the validity of all requests, and if a request is invalid, a local boolean `tampered` is set to true. When the button is pressed, the registrar removes all requests except the ones that have been received within the monitor time. Then it enters TEA mode. It continues to listen to probe requests, but now a response is sent immediately. It will remain in TEA mode for the duraction of the walk time. After that, if the `tampered` variable is true, he will enter the `TAMPERED` state. Otherwise, he will check the amount of enrollees detected. If a single enrollee was detected, he will enter the state `PAIR`, indicating that the registrar

is willing to run the WPS registration protocol with the Diffie-Hellman key of that enrollee. If zero or more than one enrollees were found, the registrar will move to `NO_ENROLLEES` and `SESSION_OVERLAP` respectively.

5.2.4 Adversary

The adversary can edit the payload of probe requests and responses, and he can impersonate an enrollee or registrar by sending out probe requests and responses. Each capability can be turned off by configuring boolean variables. When the enrollee sends a probe request, the synchronization channel `adversaryProbeRequest` is used. This allows the adversary to edit the payload. Since this models a TEA, whenever the adversary edits the data he must set the tampered field to true. The adversary now uses the `probeRequest` synchronization channel to notify the registrars of this probe request. For probe responses the channel `adversaryProbeResponse` is used, and after editing the data the adversary uses the `probeResponse` channel to notify the enrollees of the response.

6 Model checking

Each model has been checked for desired properties. The model checking results of the TEA model are presented in section 6.1, and the TEP model checking results in section 6.2. The results will be evaluated in 6.3.

6.1 TEA model checking results

The following properties have been checked on the TEA model:

1. The receiver always detects a TEA being sent
2. Without an adversary interfering, the TEA will be delivered correctly
3. Whenever an adversary tampers with the TEA it will be detected by the receiver

These properties are tested using the following queries:

1. `A<> Receiver1.Valid || Receiver1.Tampered`
2. `(Adversary1.finished && !Adversary1.tampered) --> Receiver1.Valid`
3. `Adversary1.tampered --> Receiver1.Tampered`

6.1.1 Model checking parameters

The complexity of the model is determined by multiple parameters. The length of a hash has a huge effect on the model complexity. A hash length of 16 bits has been chosen to prevent exceeding the memory limitations. The receiver will poll the medium for energy twice every sensing window, and multiplies the numerators of the fractional occupancies by 50. A higher polling frequency and a higher factor would be closer to the reality, but that introduces overflowing integers.

6.1.2 Results without adversary

In this section the queries are checked with the adversary set to inactive.

Query	Desired	Result
`A<> Receiver1.Valid || Receiver1.Tampered`	satisfied	satisfied
`(Adversary1.finished && !Adversary1.tampered) --> Receiver1.Valid`	satisfied	satisfied
`Adversary1.tampered --> Receiver1.Tampered`	satisfied	satisfied

The results show without an adversary the TEA will be detected and the TEA will be received correctly. The third query is trivial, since the adversary is inactive `Adversary1.tampered` is always false, and *ex falso quodlibet.*

6.1.3 Results with adversary

Now the queries are verified using an adversary that may edit the content of messages but may not send its own messages.

Query	Desired	Result
A<> Receiver1.Valid || Receiver1.Tampered	satisfied	satisfied
(Adversary1.finished && !Adversary1.tampered) -- > Receiver1.Valid	satisfied	satisfied
Adversary1.tampered -- > Receiver1.Tampered	satisfied	satisfied

The TEA is still detected, and the receiver will detect the tampering.

The queries are also verified using an adversary that may transmit during silent slots but may not edit the content of other messages.

Query	Desired	Result
A<> Receiver1.Valid || Receiver1.Tampered	satisfied	satisfied
(Adversary1.finished && !Adversary1.tampered) -- > Receiver1.Valid	satisfied	satisfied
Adversary1.tampered -- > Receiver1.Tampered	satisfied	satisfied

Again, all desired properties hold.

Now the queries will be verified using the full adversary, i.e. the adversary may edit the content of messages and transmit during silent slots.

Query	Desired	Result
A<> Receiver1.Valid || Receiver1.Tampered	satisfied	satisfied
(Adversary1.finished && !Adversary1.tampered) -- > Receiver1.Valid	satisfied	satisfied
Adversary1.tampered -- > Receiver1.Tampered	satisfied	satisfied

The full adversary cannot fool the receiver, any tampering is detected. The TEA model satisfies all three desired properties.

6.2 TEP model checking results

The TEP model is tested on multiple properties. The first property is: When there is one enrollee, one registrar and no adversary, they will always pair. In UPPAAL this translates to

- `A<> Enrollee1.PAIR && Enrollee1.registrarId[0] == 13`
- `A<> Registrar1.PAIR && Registrar1.enrolleeId[0] == 5`

where 13 is the Diffie-Hellman key of the registrar and 5 the Diffie-Hellman key of the enrollee. These queries are tested with the adversary configured to do nothing at all.

`A<> Enrollee1.PAIR && Enrollee1.registrarId[0] == 13`	satisfied	satisfied
`A<> Registrar1.PAIR && Registrar1.enrolleeId[0] == 5`	satisfied	satisfied

These resulst show that the model operates as expected, in all possible scenarios the enrollee and registrar will eventually pair using the correct keys.

Another property that is tested is: An enrollee or registrar will never attempt to pair using the wrong key. This means that an MITM is not possible. This property can be checked in Uppaal using the following queries, with the adversary fully enabled.

- `A<> Enrollee1.PAIR imply Enrollee1.registrarId[0] == 13`
- `A<> Registrar1.PAIR imply Registrar1.enrolleeId[0] == 5`

where 13 is the Diffie-Hellman key of the registrar and 5 the Diffie-Hellman key of the enrollee.

`A<> Enrollee1.PAIR imply Enrollee1.registrarId[0] == 13`	satisfied	satisfied
`A<> Registrar1.PAIR imply Registrar1.enrolleeId[0] == 5`	satisfied	satisfied

Both queries are satisfied, so an adversary cannot convince an enrollee or registrar to pair with it.

6.3 Evaluation of results

Three desired properties have been checked for the TEA model: A TEA is always detected, a TEA will be delivered correctly if there is no adversary interfering, and whenever an adversary tampers with the sending of a TEA the tampering will be detected. All three properties hold with every configuration of adversary. The TEP model assumes TEAs to be working as intended. This assumption is correct since the desired properties hold in the TEA model. Two properties have been checked for the TEP model: without an adversary the enrollee and registrar always pair, and an enrollee or registrar will never attempt to pair using the wrong key. Both properties hold, which means that an adversary cannot successfully impersonate a registrar to an enrollee or successfully impersonate an enrollee to a registrar. Therefore MITM attacks are impossible in the model.

The Uppaal verifier uses a 32-bit application, which can only address 4GiB of memory due to the 32-bit memory addressing. These results are less valueable because the TEA model had to be simplified in order to keep the memory usage under this limit. The attacker cannot transmit whenever he wants, but only specified timeframes in silent slots. A 64-bit version of Uppaal is in

development but is not stable yet. Due to the lack of float support in UPPAAL fractional occupancies had to be rounded after multiplying with a factor and stored as integers. Because there are no unsigned integers or longs, this factor had to be small to avoid overflowing integers. This makes it impossible to accurately store fractions.

Another measure to decrease the model complexity is splitting TEP and TEA model. Protocol composition might bring new vulnerabilities that are not checked when testing the different components individually.

7 Conclusion

Tamper-evident pairing aims to prevent man-in-the-middle attacks by using a new primitive, the tamper-evident announcement. An attacker must not be able to hide a TEA, and if an attacker tampers with the TEA, it must be detectable. An UPPAAL TEA model has been designed to check these properties. All of the desired properties hold in the model. An UPPAAL TEP model has been designed assuming TEAs to be working correctly. The model checking of the TEP model showed that without an adversary interfering, the enrollee and registrar always pair. When an adversary interferes with the protocol, the tampering is evident and the enrollee and registrar cannot be fooled to pair with the adversary. The results of the TEA model make the assumption in the TEP model valid, and therefore we can conclude that man-in-the-middle attacks are not possible in our model.

The TEP model assumes the TEA model checking results are correct, but the TEA model is simplified to lower the model complexity and memory usage for model checking. This leaves room for future work. When UPPAAL has a stable 64-bit version, these models could be configured to be more accurate at the cost of a higher memory usage. The TEP and TEA model could be merged to a single model because this might bring new vulnerabilities. Using a model checker with float support would allow one to accurately store fractions and create a model that is closer to the reality.

Another possible direction of future work is to formally prove the security of TEP using a theorem prover. This will be attempted within the Go-Green project.

8 References

Peer-reviewed literature

[1] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *4th ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal: a tool suite for automatic verification of real-time systems. *Lecture Notes in Computer Science*, 1066:232–243, 1996.

[4] R. Corin, S. Etalle, P. Hartel, and A. Mader. Timed model checking of security protocols. *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 23–32, 2004.

[5] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, pages 644–654, 1976.

[6] Whitfield Diffie, Paul C. Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, codes and cryptography*, 2:107–125, 1992.

[7] D. Dolev and A. C. Yao. On the security of public key protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, SFCS '81, pages 350–357. IEEE Computer Society, 1981.

[8] S. Gollakota, N. Ahmed, N. Zeldovich, and D. Katabi. Secure in-band wireless pairing. *Proceedings of the USENIX conference on Security*, 2011.

[9] Tzipora Halevi and Nitesh Saxena. On pairing constrained wireless devices based on secrecy of auxiliary channels: the case of acoustic eavesdropping. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 97–108. ACM, 2010.

[10] Ronald Kainda, Ivan Flechais, and A. W. Roscoe. Usability and security of out-of-band channels in secure device pairing protocols. *Proceedings of the 5th Symposium on Usable Privacy and Security*, 2009.

[11] Cynthia Kuo, Jesse Walker, and Adrian Perrig. Low-cost manufacturing, usability, and security: an analysis of bluetooth simple pairing and wi-fi protected setup. In *Proceedings of the 11th International Conference on Financial cryptography and 1st International conference on Usable Security*, FC'07/USEC'07, pages 325–340. Springer-Verlag, 2007.

[12] R Kusters and T Truderung. Using proverif to analyze protocols with diffie-hellman exponentiation. In *22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 157–171. 2009.

[13] K.S. McCurley. The discrete logarithm problem. *Cryptology and Computational Number Theory*, pages 49–74, 1990.

[14] Mehmet Orgun and Wanli Ma. An overview of temporal and modal logic programming. In *Temporal Logic*, volume 827 of *Lecture Notes in Computer Science*, pages 445–479. 1994.

[15] S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Automated Deduction-CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin / Heidelberg, 1992.

[16] Kasper B. Rasmussen, Claude Castelluccia, Thomas Heydt-benjamin, and Srdjan Capkun. Proximity-based access control for implantable medical devices. In *In CCS*, 2009.

[17] Andrew Tanenbaum. *Computer Networks*. Pearson, fourth edition, 2003.

Non-peer-reviewed literature

[18] Go-green project proposal. `http://www.agentschapnl.nl/content/project-gogreen`.

[19] Wi-Fi Alliance. Wi-fi protected setup specification. 2006.

[20] S. Viehböck. Brute forcing wi-fi protected setup. `http://sviehb.wordpress.com/2011/12/27/wi-fi-protected-setup-pin-brute-force-vulnerability`, 2011.

Appendices

A TEA model

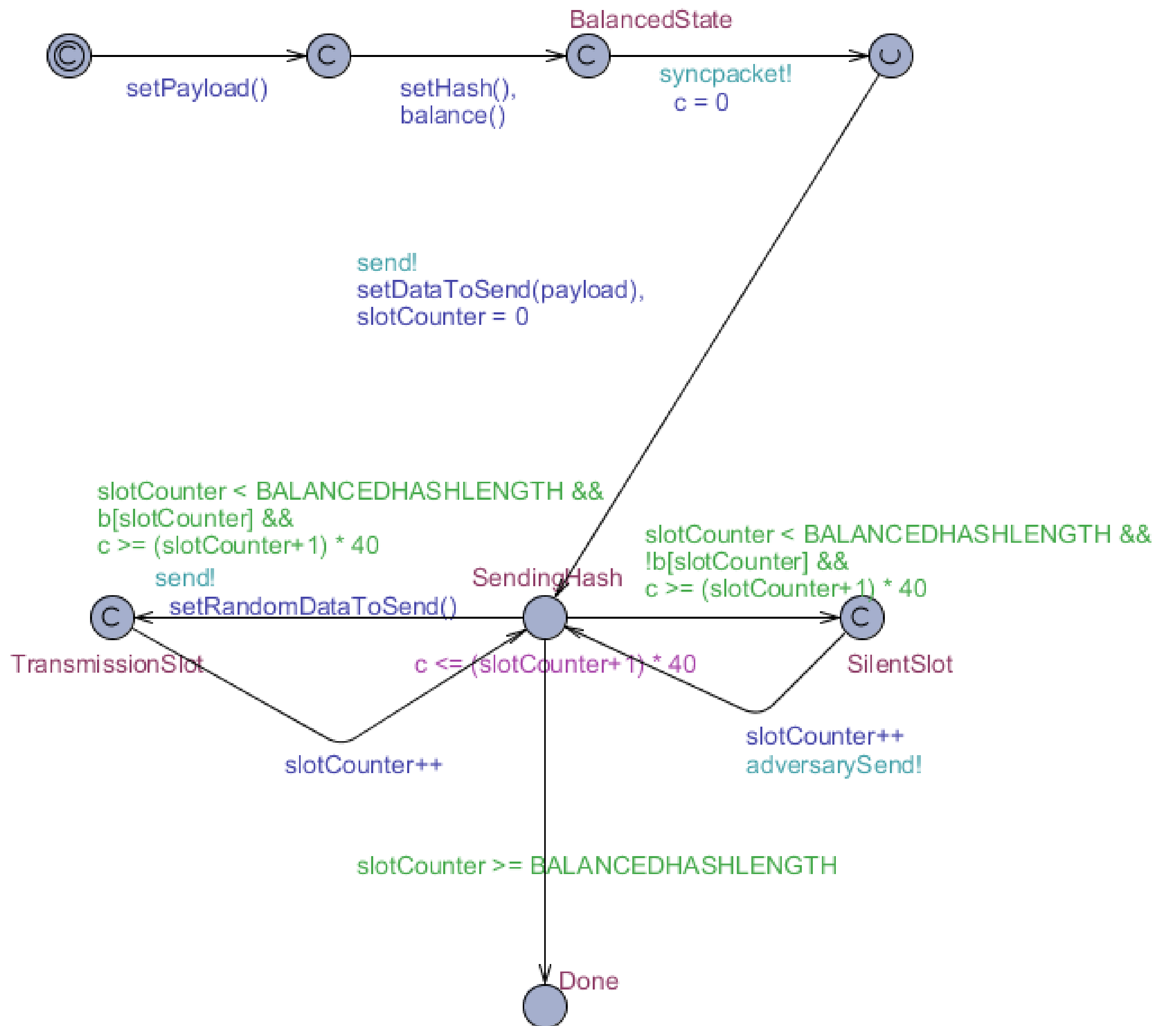
A.1 TEA model global declarations

```
//Global declarations
const int PAYLOADLENGTH = 16;
const int HASHLENGTH = 16;
const int LOGHASHLENGTH = 4;
const int BALANCEDHASHLENGTH = HASHLENGTH+2*LOGHASHLENGTH+2;
const int SLOT_DATA_LENGTH = 16;
const int SLOT_DURATION = 40;

// adjust the capabilities of the adversary
const bool adversaryEditData = true; // lets the adversary edit data in the air
const bool adversaryTransmit = true; // lets the adversary occupy the medium

//define custom types
typedef bool payload[PAYLOADLENGTH];
typedef bool hash[HASHLENGTH];
typedef bool balancedHash[BALANCEDHASHLENGTH];


broadcast chan syncpacket;
chan send;
broadcast chan transmit;
chan captureEffect;
chan adversarySend;
bool dataToSend[SLOT_DATA_LENGTH];
bool dataInAir[SLOT_DATA_LENGTH];

bool mediumBusy;
bool adversaryTransmitting;

// returns true iff the medium is busy
bool isMediumBusy() {
    return mediumBusy || adversaryTransmitting;
}

void setDataToSend(bool data[PAYLOADLENGTH]) {
    int i;
    for(i = 0; i < SLOT_DATA_LENGTH; i++) {
        dataToSend[i] = data[i];
    }
}

// calculates the hash of given payload
void getHash(bool p[PAYLOADLENGTH], hash& h) {
```

```
    int i;
    for(i = 0; i < HASHLENGTH; i++) {
        h[i] = p[i];
    }
}

//returns a to the power b
int pow(int a, int b) {
    int ret = 1;
    for(b;b>0; b--) {
        ret = ret * a;
    }
    return ret;
}

//returns the amount of ones minus the amount of zeros
int[-HASHLENGTH,HASHLENGTH] bitsDifference(hash input)
{
    int[-HASHLENGTH,HASHLENGTH] difference = 0;
    int j;
    for(j = 0; j < HASHLENGTH; j++) {
        if(input[j]) {
            difference++;
        }
        else {
            difference--;
        }
    }
    return difference;
}

//concatenates the manchaster encoding of the binary representation of i to resultArray
void intToBoolArray(balancedHash &resultArray, int i) {
    int j;
    for(j = LOGHASHLENGTH-1; j >= 0; j--) {
        if(i >= pow(2,j)) {
            i = i - pow(2, j);
            resultArray[HASHLENGTH+(LOGHASHLENGTH-1-j)*2] = true;
            resultArray[HASHLENGTH+(LOGHASHLENGTH-1-j)*2+1] = false;
        }
        else {
            resultArray[HASHLENGTH+(LOGHASHLENGTH-1-j)*2] = false;
            resultArray[HASHLENGTH+(LOGHASHLENGTH-1-j)*2+1] = true;
        }
    }
}

//Balances the number of trues and falses in a bool array by flipping all
//elements before index i
//returns i
```

```
int balanceBits(hash &input) {
    int i = 0;
    int d = bitsDifference(input);
    while(d != 0) {
        input[i] = !input[i];
        i++;
        d = bitsDifference(input);
    }
    return i;
}

//takes a hash and runs the bit-balancing algorithm.
//the result is stored on result
void bitBalancingAlgorithm(hash &input, balancedHash &result) {
    int i, j;
    i = balanceBits(input);

    //copy hash from input to output
    for(j = 0; j < HASHLENGTH; j++) {
        result[j] = input[j];
    }

    intToBoolArray(result, i);
}
```

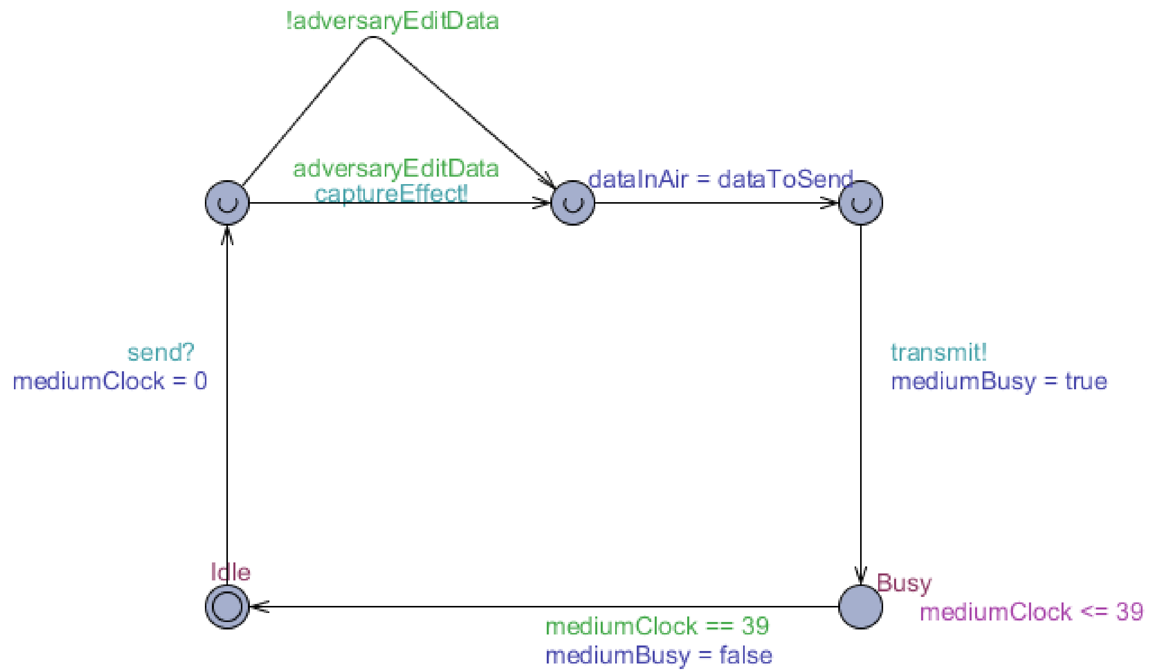
A.2 TEA model sender template




```
bool payload[PAYLOADLENGTH];
hash h;
balancedHash b;
int slotCounter;
clock c;

void setPayload() {
    int i;
    for(i = 0; i < PAYLOADLENGTH; i++) {
        payload[i] = i%3 == 0;
    }
}
```

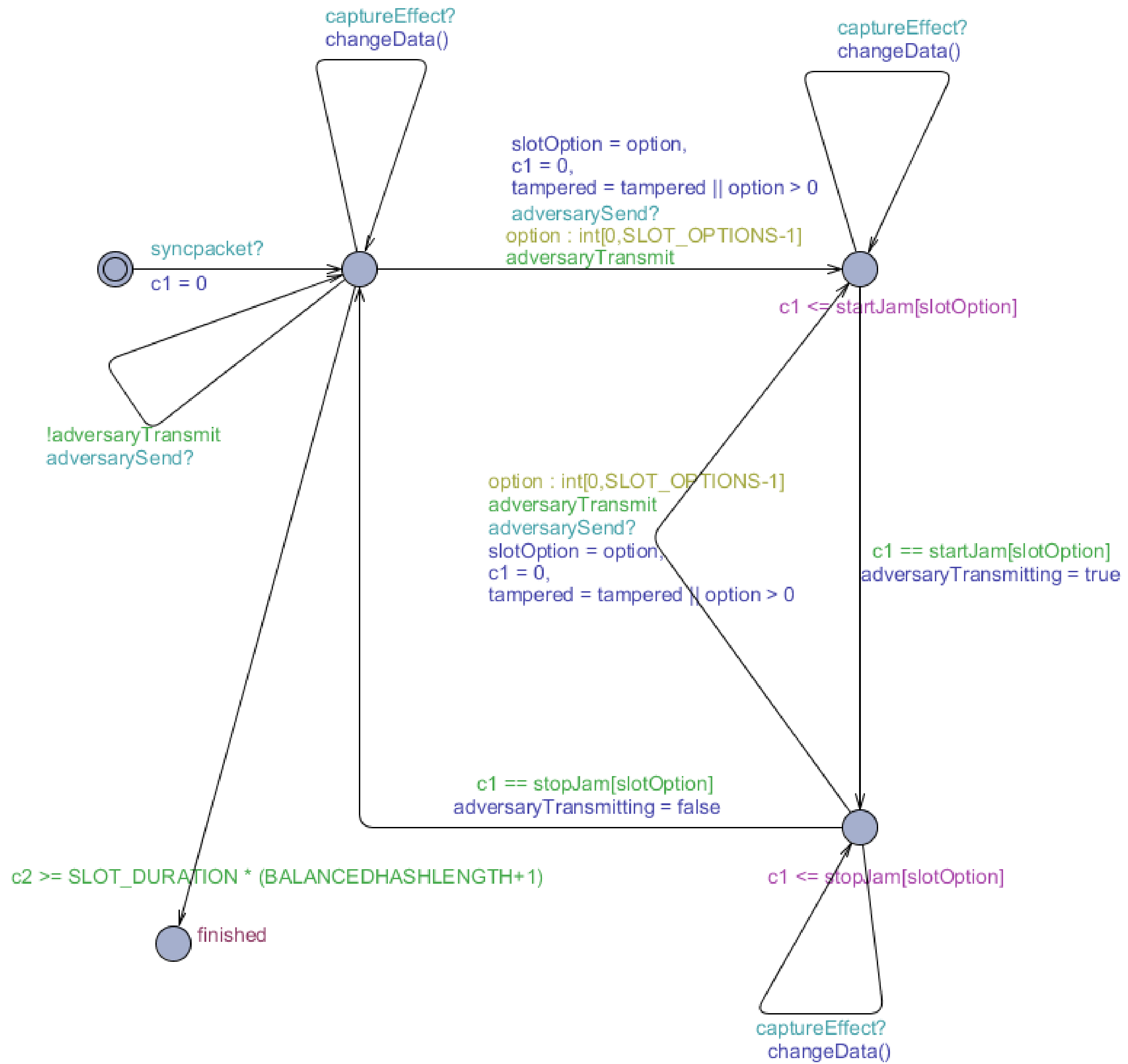
```
void setRandomDataToSend() {
    int i;
    bool data[PAYLOADLENGTH];
    for(i = 0; i < SLOT_DATA_LENGTH; i++) {
        data[i] = true;
    }

    setDataToSend(data);
}

void setHash() {
    getHash(payload, h);
}

void balance() {
    int i, j;
    bitBalancingAlgorithm(h, b);
}
```

A.3 TEA model wireless medium template




```
clock mediumClock;
```

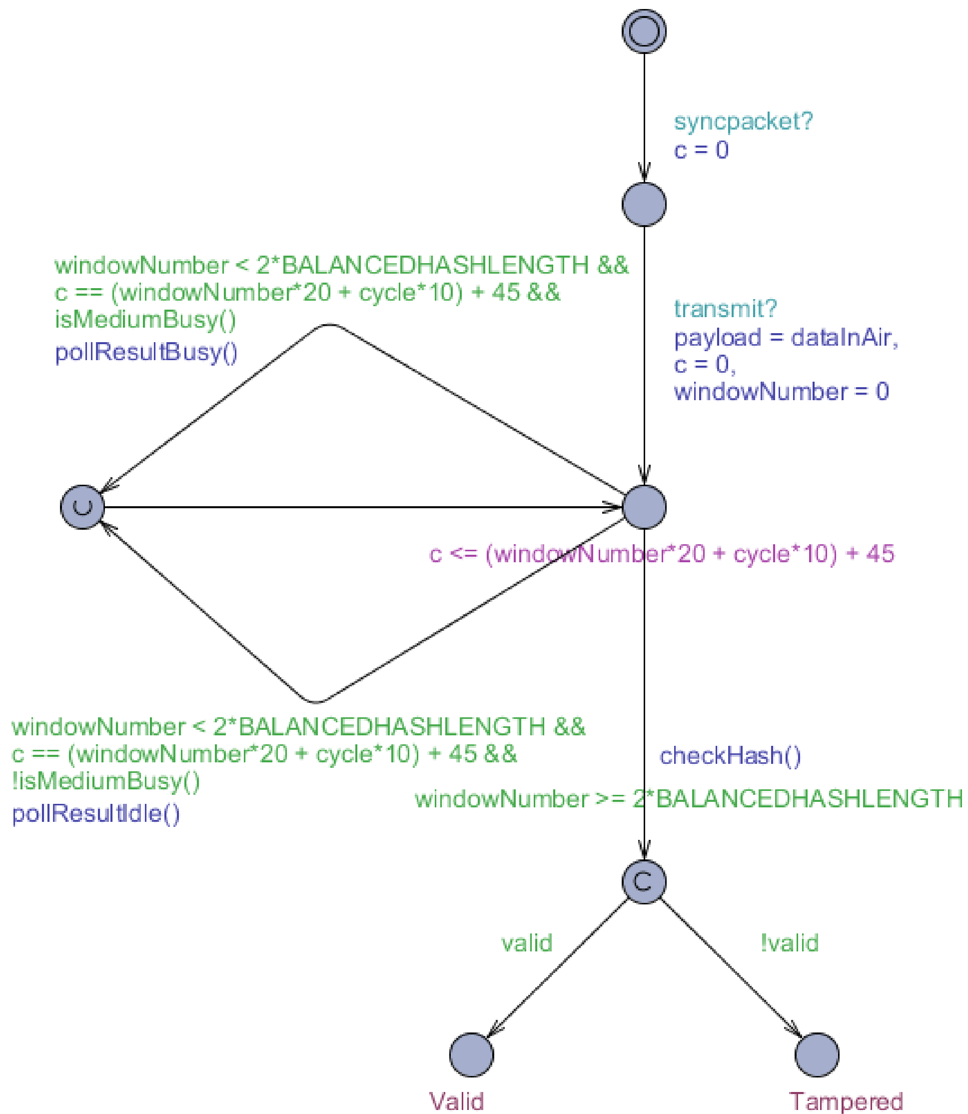
A.4 TEA model adversary template




```
const int SLOT_OPTIONS = 2;
clock c1;
clock c2;

int startJam[SLOT_OPTIONS] = {0, 0};
int stopJam[SLOT_OPTIONS] = {0, 39};
int slotOption;
```

```
bool tampered = false;

void setDataToSend() {
    int i;
    for(i = 0; i < SLOT_DATA_LENGTH; i++) {
        dataToSend[i] = true;
    }
}

void changeData() {
    if(!adversaryEditData) {
        return;
    }
    else {
        setDataToSend();
        tampered = true;
    }
}
```

A.5 TEA model receiver template




```
clock c;

bool payload[SLOT_DATA_LENGTH];
hash h;
int windowNumber = 0;
balancedHash b;
bool valid;

int rxclr = 0;
```

```
int cycle = 0;
int fractionalOccupancy[2*BALANCEDHASHLENGTH];
const int FRACTIONAL_OCCUPANCY_THRESHOLD = 25;
int var1;
int var2;

// returns round(a/b)
int roundedIntDivision(int a, int b) {
    int mod;
    int ret = a/b;
    mod = a%b;
    if(2*mod >= b) {
        ret++;
    }
    return ret;
}

void setPollResult(bool result) {
    cycle++;
    if(result) {
        rxclr++;
    }

    if(cycle == 2) {
        //the sensing window is complete, store result
        fractionalOccupancy[windowNumber] = roundedIntDivision(50*rxclr, cycle);
        windowNumber++;

        //reset registers
        rxclr = 0;
        cycle = 0;
    }
}

void pollResultBusy() {
    setPollResult(true);
}

void pollResultIdle() {
    setPollResult(false);
}

void setHash() {
    bool p[PAYLOADLENGTH];
    int i;
    for(i = 0; i < PAYLOADLENGTH; i++) {
        p[i] = payload[i];
    }
    getHash(p, h);
}
```

```
// calculate the average of all even or odd elements
// of the fractionalOccupancy array
int average(bool odd) {
    int i;
    int sum = 0;
    for(i = 0; i < BALANCEDHASHLENGTH; i++) {
        sum += fractionalOccupancy[2*i+odd];
    }

    return roundedIntDivision(sum, BALANCEDHASHLENGTH);
}

// calculate the variance of the even or odd elements
// of the fractionalOccupancy array
int variance(bool odd) {
    int i;
    int sum = 0;
    int average = average(odd);

    for(i = 0; i < BALANCEDHASHLENGTH; i++) {
        sum += pow(average-fractionalOccupancy[2*i+odd], 2);
    }

    return sum;
}

void setBalancedHash() {
    int i;
    bool odd;
    var1 = variance(true);
    var2 = variance(false);
    odd = var1 > var2;

    for(i = 0; i < BALANCEDHASHLENGTH; i++) {
        b[i] = fractionalOccupancy[2*i+odd] >= FRACTIONAL_OCCUPANCY_THRESHOLD;
    }
}

void checkHash() {
    int i;
    balancedHash correctHash;
    valid = true;

    setBalancedHash();

    //calculate the correct balanced hash
    setHash();
```

```
    bitBalancingAlgorithm(h, correctHash);

    //compare the received and correct balanced hash
    for(i = 0; valid && i < BALANCEDHASHLENGTH; i++) {
        valid = valid && correctHash[i] == b[i];
    }
}
```

A.6 TEA model system declarations

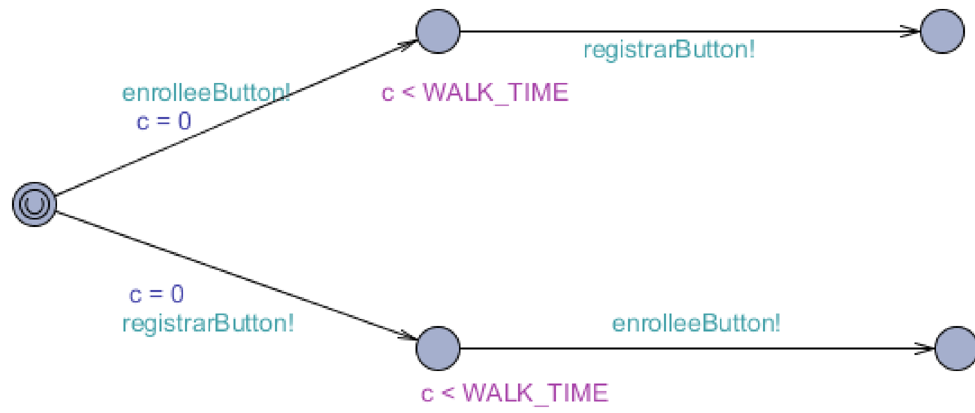
```
Sender1 = Sender();
WirelessMedium1 = WirelessMedium();
Receiver1 = Receiver();
Adversary1 = Adversary();

system Sender1, WirelessMedium1, Receiver1, Adversary1;
```

B TEP model

B.1 TEP model global declarations

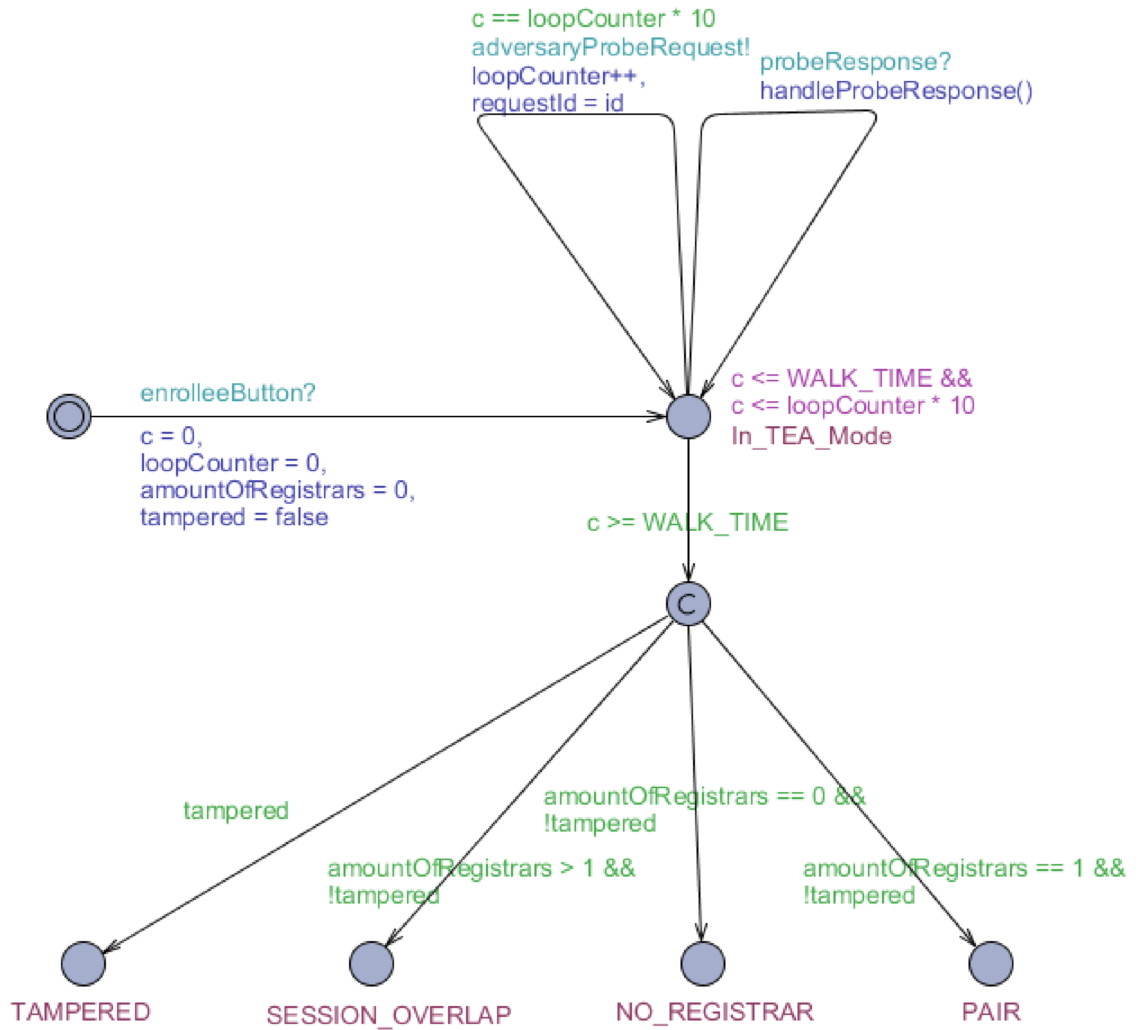
```
const int WALK_TIME = 120;
const int MONITOR_TIME = 120;
const int MAX_ARRAY_LENGTH = 5;

//synchronization channels
broadcast chan enrolleeButton;
broadcast chan registrarButton;
chan adversaryProbeRequest;
chan adversaryProbeResponse;
broadcast chan probeRequest;
broadcast chan probeResponse;

//probe request data
int requestId;
bool requestTampered;

//probe response data
int responseRegistrarId;
bool responseTampered;

//check whether element is in a with index < length
//returns the index of element in a if index < length
//returns -1 otherwise
int inList(int a[MAX_ARRAY_LENGTH], int length, int element) {
    int i;
    for(i = 0; i < length; i++) {
        if(a[i] == element) {
            return i;
        }
    }
    return -1;
}

//Remove an element from an array and shift the following
//elements to the left
void removeItemFromArray(int a[MAX_ARRAY_LENGTH], int index, int length) {
    int i;
    for(i = index; i < length-1; i++) {
        a[i] = a[i+1];
    }
}
```

B.2 TEP model user template




```
clock c;
```

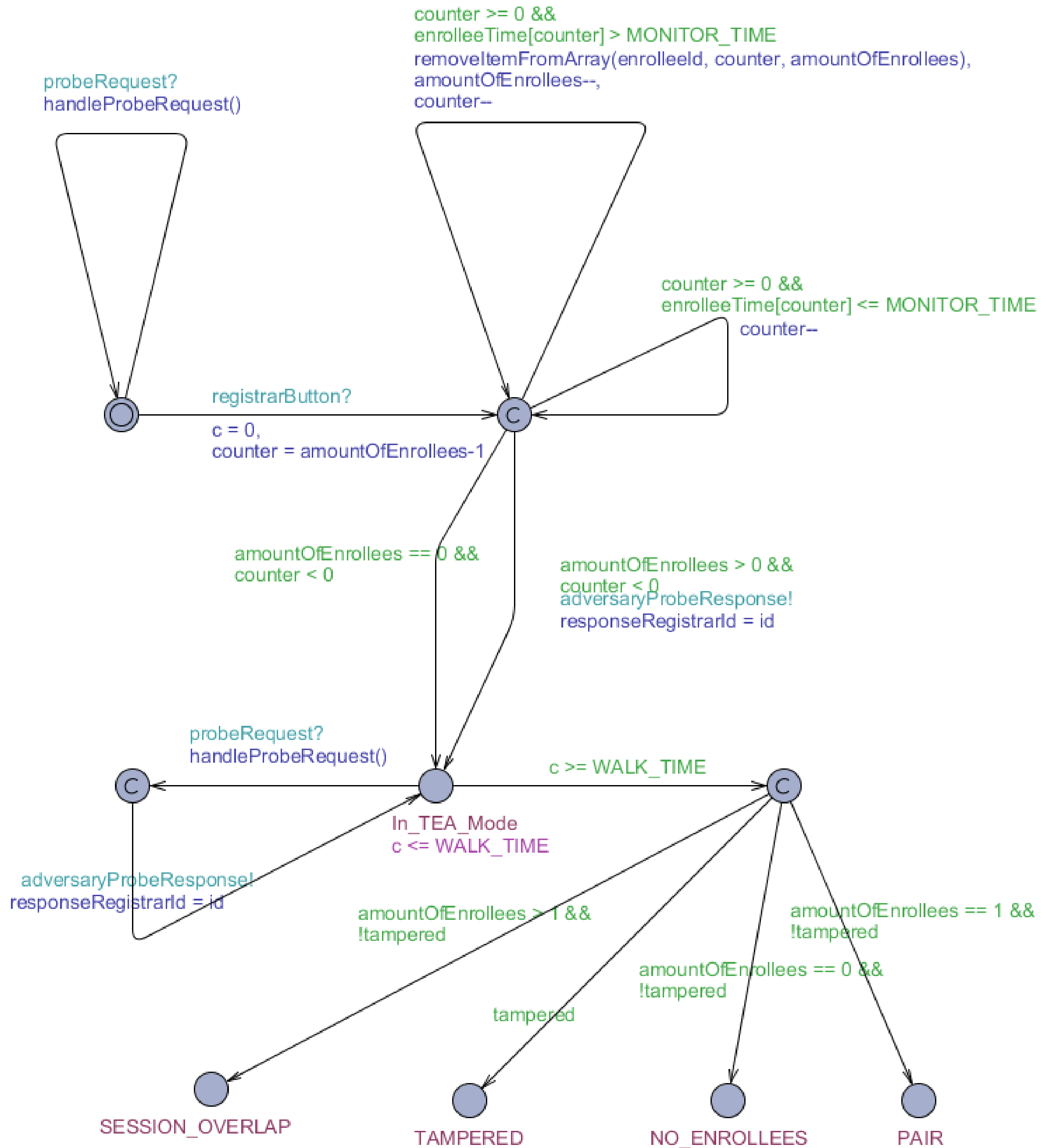
B.3 TEP model enrollee template




```
clock c;
int loopCounter;

int registrarId[MAX_ARRAY_LENGTH];
int amountOfRegistrars;
bool tampered;

void handleProbeResponse() {
    tampered = tampered || responseTampered;
    if(inList(registrarId, amountOfRegistrars, responseRegistrarId) < 0) {
        registrarId[amountOfRegistrars] = responseRegistrarId;
        amountOfRegistrars++;
    }
```

```
}
```

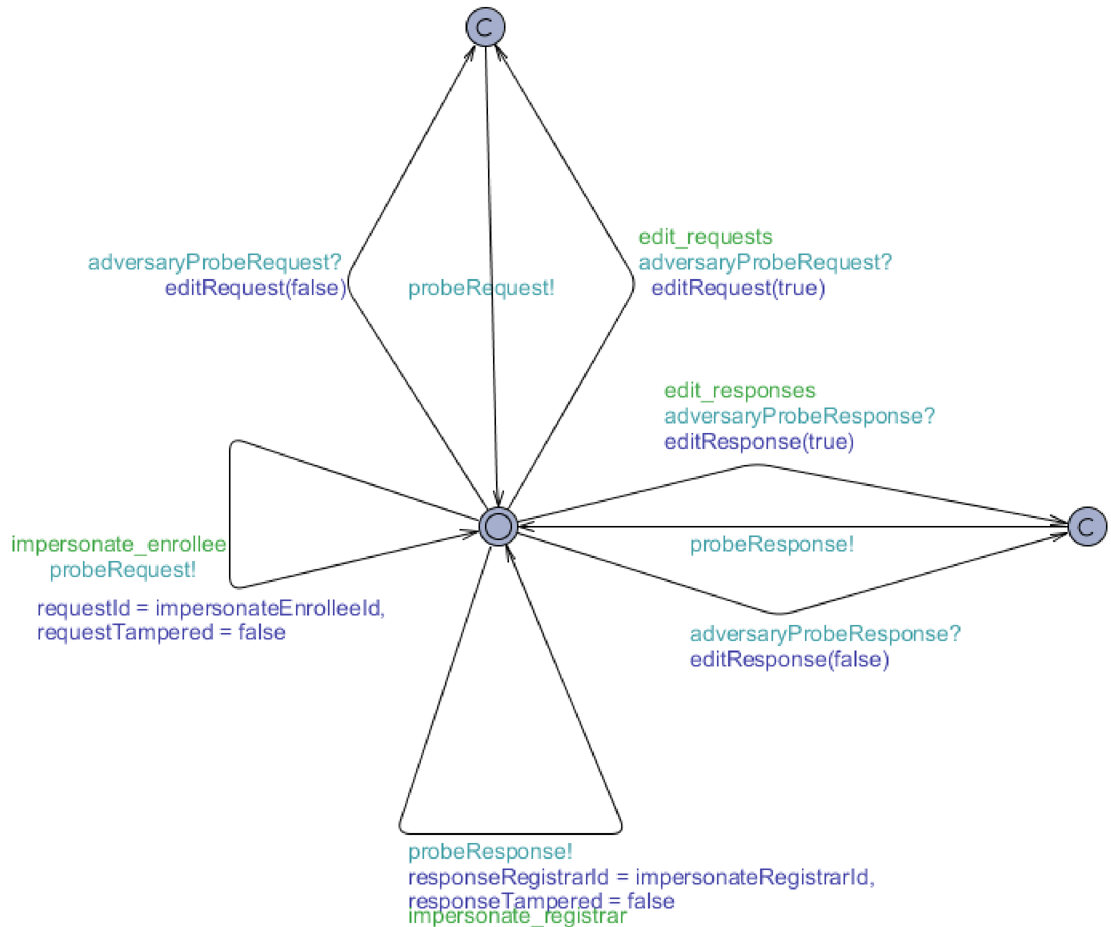
B.4 TEP model registrar template




```
clock c;

int enrolleeId[MAX_ARRAY_LENGTH];
```

```
clock enrolleeTime[MAX_ARRAY_LENGTH];
int amountOfEnrollees;
int counter;
bool mustReply;
bool tampered = false;

void handleProbeRequest() {
    int i = inList(enrolleeId, amountOfEnrollees, requestId);
    if(i >= 0) {
        enrolleeTime[i] = 0;
        //mustReply = false;
    }
    else {
        enrolleeId[amountOfEnrollees] = requestId;
        enrolleeTime[amountOfEnrollees] = 0;
        amountOfEnrollees++;
        //mustReply = true;
    }

    tampered = tampered || requestTampered;
}
```

B.5 TEP model adversary template




```
clock c;

//configure the adversary capabilities
const bool impersonate_registrar = true;
const bool impersonate_enrollee = true;
const bool edit_requests = true;
const bool edit_responses = true;

//the diffie-hellman keys used to
//impersonate a registrar and enrollee
const int registrarId = 13;
const int enrolleeId = 5;

void editRequest(bool edit) {
    if(edit) {
```

```
        requestId = impersonateEnrolleeId;
    }
    requestTampered = edit;

}

void editResponse(bool edit) {
    if(edit) {
        responseRegistrarId = impersonateRegistrarId;
    }
    responseTampered = edit;
}
```

B.6 TEP model system declarations

```
User1 = User();
Enrollee1 = Enrollee(5);
Registrar1 = Registrar(13);
Adversary1 = Adversary(10, 20);

system User1, Enrollee1, Registrar1, Adversary1;
```