

Secure Failure Detection in TrustedPals

Roberto Cortiñas¹, Felix C. Freiling², Marjan Ghajar-Azadanlou³, Alberto Lafuente¹, Mikel Larrea¹,
Lucia Draque Penso², and Iratxe Soraluze¹

¹ The University of the Basque Country, San Sebastián, Spain

² Department of Computer Science, University of Mannheim, Germany

³ Department of Computer Science, RWTH Aachen University, Germany

Abstract. We present a modular redesign of TrustedPals, a smartcard-based security framework for solving secure multiparty computation (SMC)[?]. TrustedPals allows to reduce SMC to the problem of fault-tolerant consensus between smartcards, where only process crashes and message omissions may take place. Hence, within the redesign aimed at incorporating failure detection, we investigate the problem of solving consensus in such an omission failure model augmented with failure detectors. To this end, we give novel definitions of both consensus and the class of $\diamond\mathcal{P}$ failure detectors [?] in the omission model and show how to implement $\diamond\mathcal{P}$ and have consensus in such a system with some weak synchrony assumptions. The integration of failure detection into the TrustedPals framework uses tools from privacy enhancing techniques such as message padding and dummy traffic. We close by presenting a security performance evaluation.

Keywords: failure detection, fault-tolerance, smartcards, consensus, secure multiparty computation, message padding, dummy traffic, general omission model, security performance, reliability

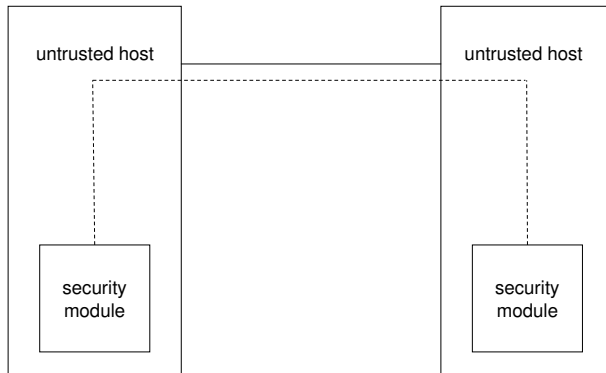


Fig. 1. Processes with tamper proof security modules.

1 Introduction

Consider a set of parties who wish to correctly compute some common function F of their local inputs, while keeping their local data as private as possible, but who do not trust each other, nor the channels by which they communicate. This is the problem of *Secure Multi-party Computation* (SMC) [17]. SMC is a very general security problem, i.e., it can be used to solve various real-life problems such as distributed voting, private bidding and auctions like Ebay, sharing of signature or decryption functions and so on. Unfortunately, solving SMC is—without extra assumptions—very expensive both in terms of communication (number of messages) and time (number of synchronous rounds).

TrustedPals [3] is a smartcard-based implementation of SMC which allows much more efficient solutions to the problem. Conceptually, TrustedPals considers a distributed system in which processes are locally equipped with tamper proof security modules (see Fig. 1). In practice, processes are implemented as a Java desktop application and security modules are realized using Java Card Technology enabled smartcards [5]. Roughly speaking, solving SMC between processes is achieved by having the security modules jointly simulate a *trusted third party* (TTP), as we now explain.

To solve SMC in the TrustedPals framework, the function F is coded as a Java function and is distributed within the network in an initial setup phase. Then processes hand their input value to their security module and the framework accomplishes the secure distribution of the input values. Finally, all security modules compute F and return the result to their process. The network of security modules sets up confidential and authenticated channels between each other and operates as a *secure overlay* within the distribution phase. Within this secure overlay, arbitrary and malicious behavior of an attacker is reduced to rather benign faulty behavior (process crashes and message omissions). TrustedPals therefore allows to reduce the security problem of SMC to a fault-tolerant synchronization problem [3], namely that of *consensus*.

To date, TrustedPals assumed a *synchronous* network setting, i.e., a setting in which all important timing parameters of the network are known and bounded. This makes TrustedPals sensitive to unforeseen variations in network delay and therefore not very suitable for deployment in networks like the Internet. In this paper, we explore how to make TrustedPals applicable in environments with less synchrony. More precisely, we explore the possibilities to implement TrustedPals in a modular fashion inspired by results in fault-tolerant distributed computing: We use an *asynchronous* consensus algorithm and encapsulate (some weak) timing assumptions within a device known as a *failure detector* [4].

The concept of a failure detector has been investigated in quite some detail in systems with merely crash faults [10]. In such systems, correct processes (i.e., processes which do not crash) must eventually

permanently suspect crashing processes. There is very little work on failure detection and consensus in message omissions environments. In fact, it is not clear what a sensible definition of a failure detector (and consensus) is in such environments because the notion of a correct process can have several different meanings (e.g., a process with no failures whatsoever or a process which just does not crash but omits messages).

1.1 Related Work

Delporte, Fauconnier and Freiling [7] were the first to investigate non-synchronous settings in the TrustedPals context. Following the approach of Chandra and Toueg [4] (and similar to this paper) they separate the trusted system into an asynchronous consensus layer and a partially synchronous failure detection layer. They assume that transient omissions are masked by a piggybacking scheme. The main difference however is that they solve a *different version of consensus* than we do: Roughly speaking, message omissions can cause processes to communicate only indirectly, i.e., some processes have to relay messages for other processes. Delporte, Fauconnier and Freiling [7] only guarantee that all processes that can communicate directly with each other solve consensus. In contrast, we allow also those processes which can only communicate indirectly to successfully participate in the consensus. As a minor difference, we focus on the class $\diamond\mathcal{P}$ of eventually perfect failure detectors whereas Delporte, Fauconnier and Freiling [7] implement the less general class Ω . Furthermore, Delporte, Fauconnier and Freiling [7] do not describe how to integrate failure detection within the TrustedPals framework: A realistic adversary who is able to selectively influence the algorithms for failure detection and consensus can cause their consensus algorithm to fail.

Recently, solving SMC *without* security modules has received some attention focusing on two-party protocols [13, 14]. In systems *with* security modules, Avoine and Vaudenay [2] examined the approach of jointly simulating a TTP. This approach was later extended by Avoine et al. [1] who show that in a system with security modules fair exchange can be reduced to a special form of consensus. They derive a solution to fair exchange in a modular way so that the agreement abstraction can be implemented in diverse manners. Benenson et al. [3] extended this idea to the general problem of SMC and showed that the use of security modules cannot improve the resilience of SMC but enables more efficient solutions for SMC problems. All these papers assume a *synchronous* network model.

Correia et al. [6] present a system which employs a real-time distributed security kernel to solve SMC. The architecture is very similar to that of TrustedPals as it also uses the notion of architectural hybridization [16]. However, the adversary model of Correia et al. [6] assumes that the attacker only has remote access to the system while TrustedPals allows the owner of a security module to be the attacker. Like other previous work [3, 2, 1] Correia et al. [6] also assume a synchronous network model at least in a part of the system.

1.2 Contributions

In this paper we present a modular redesign of TrustedPals using consensus and failure detection as modules. More specifically, we make the following technical contributions:

- We give a novel definition of $\diamond\mathcal{P}$ in the omission model and we show how to implement $\diamond\mathcal{P}$ in a system with weak synchrony assumptions in the spirit of partial synchrony [8].
- We give a novel definition of consensus in the omission model and give an algorithm which uses the class $\diamond\mathcal{P}$ to solve consensus. The algorithm is an adaptation of the classic algorithm by Chandra and Toueg [4] for the crash model.
- We integrate failure detection and consensus securely in TrustedPals by employing message padding and dummy traffic, tools known from the area of privacy enhancing techniques.
- We give a detailed security analysis of the system using the attack tree method.

1.3 Paper Outline

This paper is structured as follows: In Sect. 2 we give an overview over and motivate the system model of TrustedPals. In Sect. 3 we define and implement the failure detector $\diamond\mathcal{P}$ in the omission failure model. We then use this failure detector to solve consensus in Sect. 4. In Sect. 5 we describe how to integrate failure detection and consensus securely in the TrustedPals framework and give a brief security evaluation in Sect. 6. We conclude in Sect. 7.

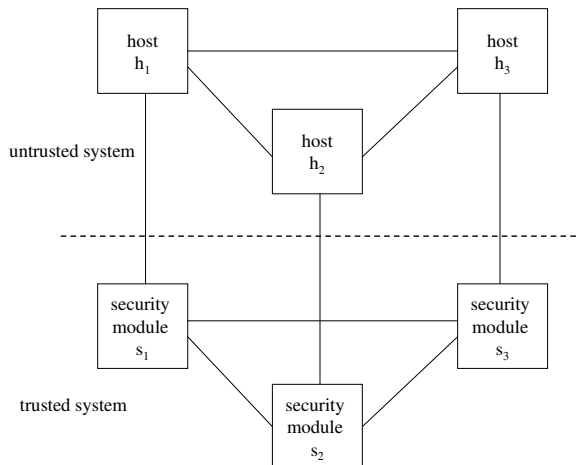


Fig. 2. The untrusted and trusted system.

2 System Model and Architecture

2.1 Untrusted and Trusted System

To be able to precisely reason about algorithms and their properties in the TrustedPals system we now formalize the system assumptions within a hybrid model, i.e., the model is divided into two parts (see Fig. 2). The upper part consists of n processes which represent the *untrusted hosts*. The lower part equally consists of n processes which represent the security modules. Because of the lack of mutual trust between untrusted hosts, we call the former part the *untrusted system*. Since the security modules trust each other we call the latter part the *trusted system*. Each host is connected to exactly one security module by a direct communication link.

Summarizing, there are two different types of processes: processes in the untrusted system and processes in the trusted system. For brevity, we will use the unqualified term *process* if the type of process is clear from the context.

Within the untrusted system each pair of hosts is connected by a pair of unidirectional communication links, one in each direction. Since the security modules also must use these links to communicate, the trusted system can be considered as an overlay network which is a network that is built on top of another network. Nodes in the overlay network can be thought of as being connected by virtual or logical links. In practice, for example, smartcards could form the overlay network which runs on top of the Internet modeled by the untrusted processes. Within the trusted system we assume the existence of a public key infrastructure, which enables two communicating parties to establish confidentiality, message integrity and user authentication without having to exchange any secret information in advance.

We assume reliable channels, i.e., every message inserted to the channel is eventually delivered at the destination. We assume no particular ordering relation on channels.

2.2 Timing Assumptions

We assume that a local clock is available to each host, but clocks are not synchronized within the network. Security modules do not have any clock, they just have a simple step counter, whereby a step consists of receiving a message from other security modules, executing a local computation, and sending a message to other security modules. Passing of time is checked by counting the number of steps executed.

Since trusted and untrusted system operate over the same physical communication channel, we assume the same timing behavior for both systems. Both systems are assumed to be *partially synchronous* meaning

that eventually bounds on all important network parameters (processing speed differences, message delivery delay) hold. The model is a variant of the partial synchrony model of Dwork, Lynch and Stockmeyer [8]. The difference is that we assume reliable channels.

We say that a message is *received timely* if it is received after the bounds on the timing parameters hold. Omission of such a message can be reliably detected using timeout-based reasoning.

2.3 Failure Assumptions

The model is hybrid because we have distinct failure assumptions for both systems. The failure model we assume in the untrusted system is the *Byzantine failure model* [12]. A Byzantine process can behave arbitrarily. In the trusted system we assume the failure model of *general omission*, which we now explain.

The concept of *omission* faults, meaning that a process drops a message either while sending (*send omission*) or while receiving it (*receive omission*), was introduced by Hadzilacos [11] and later generalized by Perry and Toueg [15]. The failure model used for the trusted system is that of *general omission*, in which processes can crash and experience either send-omissions or receive omissions. We allow the possibility of *transient* omissions, i.e., a process may temporarily drop messages and later on reliably deliver messages again.

A process (untrusted host or security module) is *correct* if it does not fail. A process is *faulty* if it is not correct. We assume a majority of processes to be correct both in the untrusted and in the trusted system. Note that a faulty security module implies a faulty host but a faulty host not necessarily implies a faulty security module.

The motivation behind this hybrid approach is that the system runs in an environment prone to attacks, but the assumptions on the security modules and the possibility to establish secure channels reduce the options of the attacker in the trusted system to attacks on the liveness of the system, i.e., destruction of the security module or interception of messages on the channel.

2.4 Classes of Processes in the Trusted System

The omission model in the trusted system implies the possibility of both transient send omissions and receive omissions. Given two processes, p and q , if a single message m sent from p to q is not delivered by q , the following question arises: has p suffered a send omission, or has q suffered a receive omission? Formally, one of the two processes is incorrect, but it is not possible to determine which one. Observe that considering both processes p and q incorrect can be too restrictive. This leads us to reconsider the different classes of processes in the omission model with respect to the common correct/incorrect classification. In particular, processes suffering a limited number of omissions, e.g., processes that do not suffer omissions with some correct process, will be considered as *good*, since they can still participate in a distributed protocol like consensus.

On the basis of this motivation, we consider the following two classes of processes:

Definition 1. A process p is *in-connected* if and only if:

- (1) p is a correct process, or
- (2) p does not crash and there exists a process q such that q is *in-connected* and all messages sent by q to p are eventually received timely by p (i.e., q does not suffer any send-omission with p , and p does not suffer any receive-omission with q).

Definition 2. A process p is *out-connected* if and only if:

- (1) p is a correct process, or
- (2) p does not crash and there exists a process q such that q is *out-connected* and all messages sent by p to q are eventually received timely by q (i.e., p does not suffer any send-omission with q , and q does not suffer any receive-omission with p).

Observe that correct processes are both in-connected and out-connected. Observe also that the definitions of in-connected and out-connected processes are recursive. Intuitively, there is a timely path with no omissions from every correct process to every in-connected process. Also, there is a timely path with

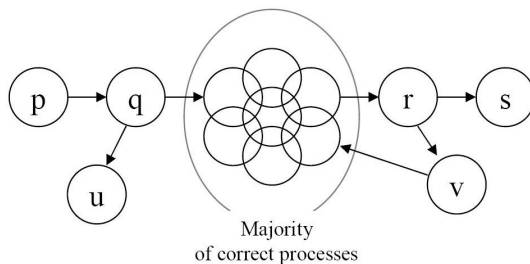


Fig. 3. Examples for classes of processes.

no omissions from every out-connected process to every correct process, and hence to every in-connected process.

Fig. 3 shows an example. In the figure, arcs represent timely links with no omissions (they are not shown for the majority of correct processes). Processes p and q are out-connected, while process s is in-connected, and processes r and v are both in-connected and out-connected. Finally, process u is neither in-connected nor out-connected.

2.5 The TrustedPals Architecture

Fig. 4 shows the layers and interfaces of the proposed modular architecture for TrustedPals. A message exchange is performed on the transport layer, which is under control of the untrusted host. The failure detector and the security mechanisms for message encryption etc. run in the TrustedPals layer. In the consensus layer runs the consensus algorithm. On the application layer, which again is under the control of the untrusted host, protocols like fair exchange operate.

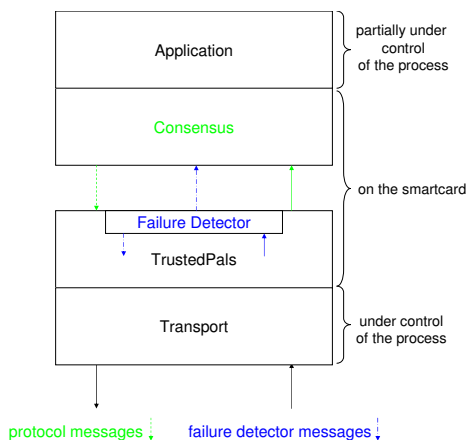


Fig. 4. The architecture of our system.

3 Failure Detection in TrustedPals

Based on the two new classes of processes defined in the previous section, we redefine now the properties that $\diamond\mathcal{P}$ must satisfy in the omission model. While the common correct/faulty classification of processes

is well addressed by means of a list of suspected processes, in the omission model we will consider two lists of processes, one for the in-connected processes and the other one for the out-connected processes. If a process p has a process q in its list of in-connected (out-connected) processes, we say that p *considers q as in-connected (out-connected)*. The $\diamond\mathcal{P}$ class of failure detectors in the omission model satisfies the following properties:

- *Strong Completeness*. Eventually every process that is not out-connected will be permanently considered as not out-connected by every in-connected process.
- *Eventual Strong Accuracy*. Eventually every process that is out-connected will be permanently considered as out-connected by every in-connected process.
- *In-connectivity*. Eventually every process that is in-connected will permanently consider itself as in-connected.

Fig. 5 presents an algorithm implementing $\diamond\mathcal{P}$. The algorithm provides to every process p a list of in-connected processes, $InConnected_p$, and another list of out-connected processes, $OutConnected_p$. For every in-connected process p , these lists will have the information required to satisfy the properties of $\diamond\mathcal{P}$. In particular, the list $OutConnected_p$ will eventually and permanently contain exactly all the out-connected processes. Regarding the $InConnected_p$ list, it will eventually and permanently contain p itself, as well as every process that is both in-connected and out-connected (hence, at least all correct processes).

In the algorithm, messages carry a sequence number in order to detect message omissions. Besides, every process p uses a matrix M_p of $n \times n$ elements. In the beginning, all processes are supposed to be correct, so every value in the matrix has 1 value. If all messages sent from a process q to a process p are received timely by p , $M_p[p][q]$ will be maintained to 1. Otherwise, process p will set $M_p[p][q]$ to 0. In this way, the matrix will have the information needed to calculate the lists of in-connected and out-connected processes.

Actually, M represents the transposed adjacency matrix of a directed graph, where the value of the element $M[p][q]$ shows if there is an arc from node q to p . We can derive from powers of the adjacency matrix if there is a path of any length between every pair of nodes. In the given algorithm, a process does not monitor itself and, as a consequence, the elements of the main diagonal of the matrix are always set to 1. Taking this into account, the n -th power of the adjacency matrix, $A_p = (M_p)^n$, gives us the information we need to obtain the sets of in-connected and out-connected processes. A process p is in-connected if it is able to receive all the messages (either directly or indirectly) from at least $\lceil \frac{(n+1)}{2} \rceil$ processes. Similarly, a process p is out-connected if at least $\lceil \frac{(n+1)}{2} \rceil$ processes are able to receive (either directly or indirectly) all the messages sent by p .

The procedure `update_In_Out_Connected_lists()` computes the lists of in-connected and out-connected processes. It is called every time a value of the matrix M_p is changed in the algorithm. The algorithm is designed to maintain the adjacency matrix M_p updated.

In Task 1 (line 14), a process p periodically sends a heartbeat message to the rest of processes. When a message is sent, the sequence number associated to the destination is incremented. Observe that the matrix M_p is sent in the heartbeat messages.

In Task 2 (line 21), if a process p does not receive the next expected message from a process q in the expected time, the value of $M_p[p][q]$ is set to 0.

In Task 3 (line 28), received messages are processed. The messages a process p receives from another process q are delivered following the sequence number $next_receive_p[q]$. Every process p has a buffer for every other process q to store unordered messages received from q . If p receives a message from q with a sequence number different from the expected one, this message is inserted in $Buffer_p[q]$ and the message is not delivered yet (line 42). A message is delivered when it is the next expected message, either because it has been just received (line 30) or it is inside the buffer (line 33). If the delivered message was in the buffer, it is removed from there. Having delivered the next expected message from a process q , if the buffer is empty it means that there is no message left from q , so $M_p[p][q]$ is set to 1. This way, process p fills its corresponding row in the matrix indicating if all the messages it expected from every other process have been received timely.

The procedure `deliver_next_message()` is used to update the adjacency matrix M_p using the information carried by the message. In the procedure, process p copies into M_p the row q of the matrix M_q received from q . This way, p learns about q 's input connectivity. With respect to every other process u , a mechanism

based on version numbers is used to avoid copying old information about u 's input connectivity. Process p will only copy into M_p the row u of M_q if its version number is higher.

```

(1) Procedure main()
(2)    $InConnected_p \leftarrow \Pi$ 
(3)    $OutConnected_p \leftarrow \Pi$ 
(4)   forall  $q \in \Pi - \{p\}$  do
(5)      $\Delta_p(q) \leftarrow$  default time-out interval       $\{\Delta_p(q)$  denotes the duration of  $p$ 's time-out interval for  $q\}$ 
(6)      $next\_send_p[q] \leftarrow 1$                      $\{\text{sequence number of the next message sent to } q\}$ 
(7)      $next\_receive_p[q] \leftarrow 1$                  $\{\text{sequence number of the next message expected from } q\}$ 
(8)      $Buffer_p[q] \leftarrow \emptyset$ 
(9)   forall  $q \in \Pi$  do
(10)    forall  $u \in \Pi$  do
(11)       $M_p[q][u] \leftarrow 1$                          $\{M_p[q][u] = 0$  means that  $q$  has not received at least one message from  $u\}$ 
(12)     $Version_p[q] \leftarrow 0$                         $\{Version_p$  contains the version number for every row of  $M_p\}$ 
(13)    $UpdateVersion \leftarrow \text{false}$ 
(14)   || Task 1: repeat periodically
(15)     if  $UpdateVersion$  then                           $\{p$ 's row has changed $\}$ 
(16)        $Version_p[p] \leftarrow Version_p[p] + 1$ 
(17)        $UpdateVersion \leftarrow \text{false}$ 
(18)     forall  $q \in \Pi - \{p\}$  do
(19)        $\text{send}(ALIVE, p, next\_send_p[q], M_p, Version_p)$  to  $q$        $\{\text{sends a heartbeat}\}$ 
(20)        $next\_send_p[q] \leftarrow next\_send_p[q] + 1$                  $\{p$  updates its sequence number for  $q\}$ 
(21)   || Task 2: repeat periodically
(22)     if  $\left( p \text{ did not receive } (ALIVE, q, next\_receive_p[q], M_q, Version_q) \right.$   $\left. \text{from } q \neq p \text{ during the last } \Delta_p(q) \text{ ticks of } p$ 's clock } \right) then  $\{\text{the next message in the sequence has not been received timely}\}$ 
(23)        $\Delta_p(q) \leftarrow \Delta_p(q) + 1$ 
(24)       if  $M_p[p][q] = 1$  then
(25)          $M_p[p][q] \leftarrow 0$ 
(26)          $UpdateVersion \leftarrow \text{true}$                  $\{\text{the potential omission is reflected in } M_p\}$ 
(27)         call  $\text{update\_In\_Out\_Connected\_lists}()$ 
(28)   || Task 3: when receive  $(ALIVE, q, c, M_q, Version_q)$  for some  $q$ 
(29)     if  $c = next\_receive_p[q]$  then                       $\{\text{it is the next message expected from } q\}$ 
(30)       call  $\text{deliver\_next\_message}(q, M_q, Version_q)$        $\{\text{the message is delivered}\}$ 
(31)        $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$ 
(32)       while  $(ALIVE, q, next\_receive_p[q], M_q, Version_q) \in Buffer_p[q]$  do
(33)         call  $\text{deliver\_next\_message}(q, M_q, Version_q)$ 
(34)          $\text{remove}(ALIVE, q, M_q, next\_receive_p[q], Version_q)$  from  $Buffer_p[q]$ 
(35)          $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$ 
(36)       if  $Buffer_p[q] = \emptyset$  then
(37)          $M_p[p][q] \leftarrow 1$ 
(38)          $UpdateVersion \leftarrow \text{true}$                  $\{\text{so far } p \text{ has received all messages from } q\}$ 
(39)       if  $M_p$  has changed then
(40)         call  $\text{update\_In\_Out\_Connected\_lists}()$ 
(41)     else
(42)        $\text{insert}(ALIVE, q, c, M_q, Version_q)$  into  $Buffer_p[q]$ 

```

Fig. 5. $\diamond\mathcal{P}$ in the omission model: main algorithm.

Correctness Proof

Lemma 1. $\forall p, q \in \Pi$, if neither p nor q crashes and there is no message omission from q to p , eventually and permanently $M_p[p][q]=1$. Otherwise, i.e., if q crashes or there is some message omission from q to p , eventually and permanently $M_p[p][q]=0$.

Proof. If all messages sent by q to p are eventually received by p , every heartbeat message sent by Task 1 of q will be eventually received by Task 3 of p . Observe that the $\text{deliver_next_message}()$ procedure does not modify $M_p[p][q]$. By Task 2 of p , $M_p[p][q]$ is set to 0 when the next expected message m from q has not been received timely by p . Eventually, on the reception of m by Task 3 of p , m will be delivered. After that,

Result: $InConnected_p$ and $OutConnected_p$ lists

```

(43) Procedure update_In_Out_Connected_lists()
(44)    $A_p \leftarrow (M_p)^n$  { $A_p$  is the  $n$ -th power of the  $M_p$  matrix}
(45)   forall  $u, v \in \Pi$  do
(46)     if  $A_p[u][v] > 0$  then
(47)        $A_p[u][v] \leftarrow 1$ 
(48)    $In \leftarrow \emptyset$ 
(49)    $Out \leftarrow \emptyset$ 
(50)   forall  $q \in \Pi$  do
(51)     if  $(\sum_{i=0}^{n-1} A_p[q][i] \geq \lceil \frac{(n+1)}{2} \rceil)$  then
(52)        $In \leftarrow In \cup \{q\}$ 
(53)     if  $(\sum_{i=0}^{n-1} A_p[i][q] \geq \lceil \frac{(n+1)}{2} \rceil)$  then
(54)        $Out \leftarrow Out \cup \{q\}$ 
(55)    $InConnected_p \leftarrow In$ 
(56)    $OutConnected_p \leftarrow Out$ 

```

Fig. 6. $\diamond\mathcal{P}$ in the omission model: procedure update_In_Out_Connected_lists().

Input: q : process from which the message has been received; M_q : q 's knowledge about the system; $Version_q$: version number of each row of M_q

Result: update of M_p matrix and $Version_p$ vector

```

(57) Procedure deliver_next_message()
(58)   forall  $v \in \Pi$  do { $q$ 's row of  $M_q$  is systematically copied into  $M_p$ }
(59)      $M_p[q][v] \leftarrow M_q[q][v]$ 
(60)   forall  $u \in \Pi - \{p, q\}$  do
(61)     if  $Version_q[u] > Version_p[u]$  then { $q$ 's information about  $u$  is more recent than  $p$ 's}
(62)       forall  $v \in \Pi$  do
(63)          $M_p[u][v] \leftarrow M_q[u][v]$ 
(64)          $Version_p[u] \leftarrow Version_q[u]$ 

```

Fig. 7. $\diamond\mathcal{P}$ in the omission model: procedure deliver_next_message().

if $Buffer_p[q]$ becomes empty, $M_p[p][q]$ will be set to 1. Observe that this will permanently happen when $\Delta_p(q)$ reaches the unknown bound on message transmission time. Since $\Delta_p(q)$ is incremented when m is not received timely, and since the communication link between q and p is eventually timely, this bound will be eventually reached, after which p will receive every $(ALIVE, q, -, -, -)$ message always before $\Delta_p(q)$ expires, and $M_p[p][q] = 1$ forever.

On the other hand, if a message m is omitted from q to p , by Task 2 of p , $M_p[p][q]$ is set to 0 because m has not been received timely by p . After that, p could receive a message l with a higher sequence number from q . This message would be inserted in $Buffer_p[q]$ because it was not the next expected message. If the next expected message m is never received, $Buffer_p[q]$ will never become empty and $M_p[p][q] = 0$ forever. Observe that if p does not receive more messages from q after the omission of m because all the messages are omitted or q has crashed, Task 3 of p will never be executed due to a reception of a message from q , and $M_p[p][q] = 0$ forever too.

Lemma 2. $\forall p, q \in \Pi$, if neither p nor q crashes and there is a path with no omission from q to p , eventually and permanently the row q of matrices M_p and M_q will be identical, i.e., $M_p[q][\] = M_q[q][\]$.

Proof. Process q has information about its own in-connectivity in the row q of its matrix $M_q[q][\]$. Every time a value of this row changes, its version number will be incremented. By Lemma 1, this information eventually stabilizes in q and its version number will be the highest associated to this row in the system. When a process r receives a message from q it will copy the row $M_q[q][\]$ into $M_r[q][\]$ if the row version number is higher, that is to say, the received information is newer. If there is no message omission from process q to process r , r will obtain the last version of the in-connectivity information of q . After updating the information about q , by Task 1 r will send this information in its own matrix to the rest of processes.

If there is a path with no omission from q to p , recursively, p will receive the latest information about q going through all the processes in the path. The version number mechanism ensures that old information about q arriving at p will not be copied into M_p . As a consequence, $M_p[q][\] = M_q[q][\]$ eventually and permanently.

Lemma 3. $\forall p, q \in \Pi$, if neither p nor q crashes and there is a path with no omission from q to p , eventually and permanently $(M_p)^n[p][q] \geq 1$.

Proof. The path from q to p will be composed of processes q, u, v, \dots, p . If this is a path with no omission, there is no omission from q to u , from u to v and so on. By Lemma 1, $M_u[u][q] = 1$, $M_v[v][u] = 1$ and so on. By Lemma 2, the rows q, u, v, \dots, p of the adjacency matrix M_p will be eventually updated with the in-connectivity information of all these processes, reflecting this path in the matrix. This way, the n -th power of the adjacency matrix will tell us that there is a path of some length from q to p , being $(M_p)^n[p][q] \geq 1$.

Lemma 4. $\forall p$ in-connected, $\forall q$ out-connected, eventually and permanently $q \in \text{OutConnected}_p$.

Proof. By definition, since process p is in-connected there is a path with no omission from some correct process to p . By definition too, since process q is out-connected there is a path with no omission from q to some correct process. Observe that all correct processes never suffer any omission, and considering that a majority of processes is correct, by Lemma 1, every correct process will have at least $\lceil \frac{(n+1)}{2} \rceil$ 1 values in its row. Even more, by Lemma 2, every correct process will update its matrix copying all the rows of the rest of correct processes, having eventually and permanently at least $\lceil \frac{(n+1)}{2} \rceil$ 1 values in its column. Considering that there is no omission among correct processes, we can derive that there is a path with no omission from q to every correct process and also that there is a path with no omission from every correct process to p . By Lemma 3, for every correct process r , $(M_r)^n[r][q] \geq 1$. Being a path with no omission from all correct processes to p , by Lemma 2, p will update its matrix from the correct processes, and $(M_p)^n[q] \geq 1$ for more than $\lceil \frac{(n+1)}{2} \rceil$ processes (at least the correct processes). As a consequence, according to the procedure `update_In_Out_Connected_lists()`, q will be permanently included in the list `OutConnectedp`.

Lemma 5. $\forall p$ in-connected, $\forall q$ not out-connected, eventually and permanently $q \notin \text{OutConnected}_p$.

Proof. Since q is not out-connected, it does not exist a correct process r such that there is a path with no omission from q to r . By Lemma 2, if p is in-connected, p will eventually and permanently know about the connectivity of every correct process r , and therefore, $(M_p)^n[r][q] = 0$. Since there is a majority of correct processes, the number of processes such that $(M_p)^n[q] \geq 1$ will always be less than $\lceil \frac{(n+1)}{2} \rceil$, so eventually and permanently p will consider q as not out-connected ($q \notin \text{OutConnected}_p$).

Lemma 6. $\forall p$ in-connected, eventually and permanently $p \in \text{InConnected}_p$.

Proof. As shown in Lemma 4, there is a path with no omission from every correct process to p . Considering that there is a majority of correct processes, the number of processes with a path with no omission to p will be at least $\lceil \frac{(n+1)}{2} \rceil$, and by Lemma 3, p will eventually and permanently consider itself as in-connected in the procedure `update_In_Out_Connected_lists()`.

Theorem 1. The algorithm of Figure 5 implements $\diamond\mathcal{P}$ in the omission model.

Proof. The strong completeness, eventual strong accuracy, and in-connectivity properties of $\diamond\mathcal{P}$ are satisfied by Lemmas 5, 4, and 6 respectively.

4 $\diamond\mathcal{P}$ -based Consensus in TrustedPals

In the *consensus* problem, every process proposes a value, and correct processes must eventually decide on some common value that has been proposed. In the crash model, every *correct process* is required to eventually decide some value. This is called the *Termination* property of consensus. In order to adapt consensus to the omission model, we argue that only the Termination property has to be redefined. This property involves now every in-connected process, since, despite they can suffer some omissions, in-connected processes are those that will be able to decide.

The properties of consensus in the omission model are the following:

- *Termination.* Every *in-connected* process eventually decides some value.

- *Integrity.* Every process decides at most once.
- *Uniform agreement.* No two processes decide differently.
- *Validity.* If a process decides v , then v was proposed by some process.

Fig. 8 presents an algorithm solving consensus using $\diamond\mathcal{P}$ in the omission model. It is an adaptation of the well-known Chandra-Toueg consensus algorithm. Instead of explaining the algorithm from scratch, we just comment on the modifications required to adapt the original algorithm:

- In Phase 2, the current coordinator waits for a majority of estimates while it considers itself as in-connected in order not to block. Only in case it receives a majority of estimates a valid estimate is sent to all. If it is not the case, the coordinator sends a *NEXT* message indicating that the current round cannot be successful.
- In Phase 3, every process p waits for the new estimate proposed by the current coordinator while p considers itself as in-connected and the coordinator as out-connected in order not to block. Also, p can receive a *NEXT* message indicating that the current round cannot be successful. In case p receives a valid estimate, it replies with a *ack* message. Otherwise, p sends a *nack* message to the current coordinator.
- In Phase 4, if the current coordinator sent a valid estimate in Phase 2, it waits for replies of out-connected processes while it considers itself as in-connected in order not to block. If a majority of processes replied with *ack*, the coordinator R-broadcasts a decide message.

When a process p sends a consensus message m to another process q , the following approach is assumed: (1) p sends m to all processes, including q , except p itself, and (2) whenever p receives for the first time a message m whose destination is another process q different from p , p forwards m to all processes (except the process from which p has received m and p itself). Clearly, this approach can take advantage of the underlying all-to-all implementation of the $\diamond\mathcal{P}$ failure detector.

Correctness Proof

We provide here a proof sketch of our adapted consensus algorithm in the omission model. First of all, observe that uniform agreement is preserved, because we keep the original mechanism based on majorities to decide on a value. Also, it is easy to see that integrity and validity are satisfied. Finally, in order to show that termination is satisfied, we first show that the algorithm does not block in any of its **wait** instructions:

- In Phase 2, if the current coordinator p is not in-connected, it will eventually stop waiting because the failure detector will eventually exclude p from $InConnected_p$. On the other hand, if p is in-connected, it will eventually receive a majority of estimates since there is a majority of correct processes in the system. Hence, no coordinator blocks forever in the **wait** instruction of Phase 2.
- In Phase 3, every process p waits for the new estimate proposed by the current coordinator or a *NEXT* message while p considers itself as in-connected and the coordinator as out-connected. Clearly, by the properties of $\diamond\mathcal{P}$ no process blocks forever in the **wait** instruction of Phase 3.
- In Phase 4, the current coordinator waits for replies of out-connected processes while it considers itself as in-connected. Again, by the properties of $\diamond\mathcal{P}$ and the fact that there is a majority of correct processes in the system, no coordinator blocks forever in the **wait** instruction of Phase 4.

By the previous, eventually some correct process c will coordinate a round in which:

- In Phase 2, c will receive a majority of estimates, because c will be permanently in $InConnected_c$ (by the properties of $\diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will send a valid estimate to all processes at the end of Phase 2.
- In Phase 3, every correct process p will receive c 's valid estimate, because p will be permanently in $InConnected_p$ and c will be permanently in $OutConnected_p$ (by the properties of $\diamond\mathcal{P}$). Hence, p will send a *ack* message to c at the end of Phase 3.
- In Phase 4, c will receive a majority of *ack* messages, because c will be permanently in $InConnected_c$ and all correct processes will be permanently in $OutConnected_c$ (by the properties of $\diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will R-broadcast the decision, and every in-connected process will eventually decide.

```

{Every process  $p$  executes the following}
(1) Procedure  $propose(v_p)$ 
(2)    $estimate_p \leftarrow v_p$  { $estimate_p$  is  $p$ 's estimate of the decision value}
(3)    $state_p \leftarrow undecided$ 
(4)    $r_p \leftarrow 0$  { $r_p$  is  $p$ 's current round number}
(5)    $ts_p \leftarrow 0$  { $ts_p$  is the last round in which  $p$  updated  $estimate_p$ , initially 0}
{Rotate through coordinators until decision is reached}
(6)   while  $state_p = undecided$  do
(7)      $r_p \leftarrow r_p + 1$ 
(8)      $c_p \leftarrow (r_p \bmod n) + 1$  { $c_p$  is the current coordinator}
(9)     Phase 1: {All processes  $p$  send  $estimate_p$  to the current coordinator}
(10)    [ send  $(p, r_p, estimate_p, ts_p)$  to  $c_p$ 
(11)
(12)    Phase 2: { The current coordinator tries to gather  $\lceil \frac{(n+1)}{2} \rceil$  estimates. If it succeeds,
    it proposes a new estimate. Otherwise, it sends a NEXT message to all }
(13)    if  $p = c_p$  then
(14)      wait until  $\left( (p \in \Pi - InConnected_p) \text{ or } \right.$ 
(15)       $\left. \text{for } \lceil \frac{(n+1)}{2} \rceil \text{ processes } q: \text{ received } (q, r_p, estimate_q, ts_q) \text{ from } q \right)$ 
(16)      if for  $\lceil \frac{(n+1)}{2} \rceil$  processes  $q$ : received  $(q, r_p, estimate_q, ts_q)$  from  $q$  then
(17)         $success_p \leftarrow TRUE$ 
(18)         $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p \text{ received } (q, r_p, estimate_q, ts_q) \text{ from } q\}$ 
(19)         $t \leftarrow$  largest  $ts_q$  such that  $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$ 
(20)         $estimate_p \leftarrow$  select one  $estimate_q$  such that  $(q, r_p, estimate_q, t) \in msgs_p[r_p]$ 
(21)        send  $(p, r_p, estimate_p)$  to all
(22)      else
(23)         $success_p \leftarrow FALSE$ 
(24)        send  $(p, r_p, NEXT)$  to all
(25)
(26)    Phase 3: {All processes wait for the new estimate proposed by the current coordinator}
(27)    wait until  $\left( (p \in \Pi - InConnected_p) \text{ or } \right.$ 
(28)     $\left. \text{received } [(c_p, r_p, estimate_{c_p}) \text{ or } (c_p, r_p, NEXT)] \text{ from } c_p \text{ or } \right.$ 
(29)     $\left. (c_p \in \Pi - OutConnected_p) \right)$ 
(30)    if received  $(c_p, r_p, estimate_{c_p})$  from  $c_p$  then
(31)       $estimate_p \leftarrow estimate_{c_p}$ 
(32)       $ts_p \leftarrow r_p$ 
(33)      send  $(p, r_p, ack)$  to  $c_p$ 
(34)    else
(35)      send  $(p, r_p, nack)$  to  $c_p$ 
(36)
(37)    Phase 4: { If the current coordinator sent a valid estimate in Phase 2, it waits for replies of
    out-connected processes while it considers itself as in-connected. If  $\lceil \frac{(n+1)}{2} \rceil$ 
    processes replied with ack, the coordinator R-broadcasts a decide message }
(38)    if  $(p = c_p)$  and  $(success_p = TRUE)$  then
(39)      wait until  $\left[ (p \in \Pi - InConnected_p) \text{ or } \right.$ 
(40)       $\left. \text{for all process } q: \left( \begin{array}{l} \text{received } (q, r_p, ack) \text{ or} \\ \text{received } (q, r_p, nack) \text{ or} \\ q \in \Pi - OutConnected_p \end{array} \right) \right]$ 
(41)      if for  $\lceil \frac{(n+1)}{2} \rceil$  processes  $q$ : received  $(q, r_p, ack)$  then
(42)        [ R-broadcast $(p, r_p, estimate_p, decide)$ 
{If  $p$  R-delivers a decide message,  $p$  decides accordingly}
(43) when R-deliver $(q, r_q, estimate_q, decide)$  do
(44)   if  $state_p = undecided$  then
(45)      $decide(estimate_q)$ 
(46)      $state_p \leftarrow decided$ 

```

Fig. 8. Solving consensus in the omission model using $\diamond\mathcal{P}$.

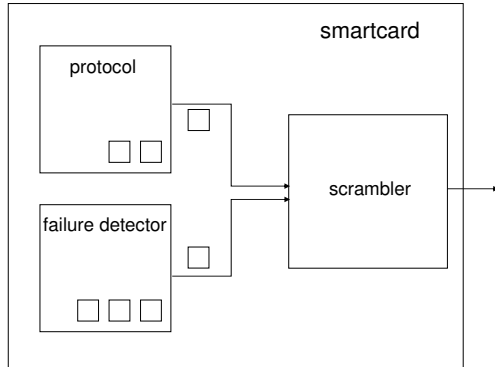


Fig. 9. Smartcard with scrambler.

5 Integrating Failure Detection and Consensus Securely

As depicted in Fig. 4, the TrustedPals layer receives messages from the consensus protocol and from the failure detector. If an untrusted host could distinguish protocol messages from failure detector messages he could intercept all former messages while leaving the latter untouched. This would result in a failure detector working properly but a consensus protocol to block forever. In order to prevent such malicious actions we piggyback the protocol messages on the failure detector messages, which are sent in regular time intervals. To make sure that the adversary can not distinguish the packets with the protocol message piggybacked from the ones without protocol message, packets will have the same size, i.e., failure detector messages are padded and protocol messages are divided into a predefined length. It might be inefficient for small messages to be padded or large packets split up in order to get a message of the desired size. However, it is necessary to find an acceptable tradeoff between security and performance such that a message size provides better security in expense of worse performance.

We assume a *scrambler* which receives the protocol and failure detector messages and outputs equal looking messages of the same size in regular time intervals (see Fig. 9). It proceeds as follows. Whenever a protocol message has to be sent, it will be piggybacked on the failure detector message. If there is no protocol message ready to be sent, the packet's payload will be filled with random bits. In order to be efficient, the predefined size of the messages sent will be kept as small as possible. If a protocol message is too big, it will be divided, using a fragmentation mechanism, and piggybacked into multiple failure detector messages. Since the protocol is asynchronous, even long delays can be tolerated as long as the failure detector works correctly.

Cryptography is applied to prevent and detect cheating and other malicious activities. We use a public key cryptosystem for encryption. Each message m in our model will be signed and then encrypted in order to reach authenticity, confidentiality, integrity, and non-repudiation.

The source and destination address are encrypted because this enables the receiver of a message to check whether the received message was intended for it or not and who the sender was. Thus, a malicious process cannot change the destination address in the header of a message from its security module and send it to an arbitrary destination without being detected. To detect a message deletion or loss, each message which is sent gets an identification number, where the fragment offset field determines the place of a particular fragment in the original message with same identification number.

As an example for the scrambler's function, consider the situation where the scrambler takes a protocol message m , whose size is three times the size of a failure detector message, from the queue of protocol messages to be sent. The scrambler divides the protocol message in three parts and assigns the next

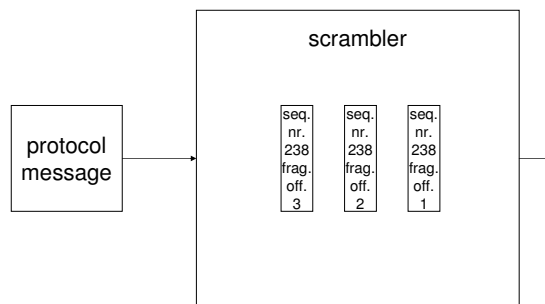


Fig. 10. Example of scrambler's function.

```

1  if (queue of protocol messages is empty) {
2      take a failure detector message
3      assign a sequence number
4      fill fragment offset field and data field with random bits
5      sign message payload
6      encrypt signature
7      add ciphertext into failure detector message payload
8      send generated message in next upcoming interval
9  }

```

Fig. 11. Part of scrambler's function in pseudocode.

available sequence number to each part. Also each part gets a fragment offset. The first message part gets the fragment offset 1, the second message part gets the fragment offset 2, and the last message part gets the fragment offset 3 (see Fig. 10). Next, the sequence number, all other fields, and the first message part all together are signed with the private key of the sender. After that, the signature is encrypted. Then, the next failure detector message is taken from the queue of failure detector messages to be sent and the encrypted message part is inserted into the failure detector message payload. Now, the first message part is ready to be sent in the next upcoming interval. The same is applied to the second and third part of the protocol message.

When the queue of protocol messages is empty, the scrambler only takes a failure detector message from the queue of failure detector messages (see Fig. 11) . Here, also a sequence number is assigned. But the fragment offset and the data field are filled with random bits. Then, the sequence number, the not set *CD* field, the not set *MF* field, the source and destination address of the message, and the random bits are taken and signed all together. Now, the signature is encrypted and added to the failure detector message payload. Then, the message can be sent in the next upcoming interval.

6 Security Evaluation

The correct execution and termination of the algorithm must be provided and all parties must have the confidence that certain objectives associated with the algorithm's security have been met. The system must provide reliable multi-party interaction under partial synchrony and subject to malicious as well as accidental faults. We evaluate if all desired security objectives are accomplished in the proposed system model. Since the failure model assumed in the untrusted system is the Byzantine failure model, malicious processes can collude, exchange information, and jointly gather knowledge to perform any kind of attack on the system. On the other hand, correct processes try to achieve safety and liveness and keep the messages exchanged secret.

Goal 1: Violate safety properties of the system

1. Cause safety properties of consensus to fail
 - (a) Violate validity property (OR)
 - i. Attack integrity
 - A. Manipulate message payload (OR)
 1. Cryptoanalyze asymmetric encryption (OR)
 2. Flip some bits (OR)
 3. Replace some blocks with previously sent message blocks
 - B. Manipulating message header
 1. Spoofing
 - (b) Violate agreement property (OR)
 - i. Attack integrity (OR)
 - ii. Delete messages (AND)
 - iii. Inject false messages (OR)
 - iv. Inject replays of previous messages
 - (c) Attack failure detector
 - i. Violate eventual strong accuracy property
 - A. Delete messages * (OR)
 - B. Attack processes' availability *

Fig. 12. Attack tree for the threat of safety properties.

To model the security threats against the system we make use of attack trees. Attack trees are conceptual diagrams of threats on systems and possible attacks to reach those threats. The reason we have chosen attack trees and not formal proofs to perform a security analysis is because verifying information flow properties is complex and different from proving safety and liveness properties

In the next section, we introduce three attack trees. The leafs of the trees are used as a basis for further discussion, where the attacks they represent are analyzed in more detail, including capabilities needed by the attacker.

6.1 Analysis

In this section we perform an analysis of the proposed system using the methodology of attack trees. We have to examine the following attacker goals:

Goal 1: Violate safety properties of the system

Goal 2: Violate liveness properties of the system

Goal 3: Violate information flow properties of the system

In the following we will give attack trees for each of these goals.

Attacks Aimed at Safety Properties of the System Figure 12 shows an attack tree for the threat of the system's safety properties. The goal is to violate the safety properties of the system. In order to violate the safety properties of the system, the attacker can cause the safety properties of the consensus algorithm to fail. This can be done either by violating the validity property of the algorithm or by violating the agreement property of the algorithm, or attacking the failure detector. The validity property can be violated by attacking the integrity of the system. The agreement property can also be violated by attacking the integrity of the system or by deleting and injecting messages. To attack the failure detector, the attacker has to violate the eventual strong accuracy property of the failure detector. Note that it can be detected if messages were corrupted in the network and corruptions are converted to omission failures.

Attacks Aimed at Liveness Properties of the System Figure 13 shows an attack tree for the threat of the system's liveness properties. The goal is to violate the liveness properties of the system. For this purpose, an attacker can violate the liveness properties of the consensus algorithm. In order to attack the liveness properties of the consensus algorithm, it either has to violate the termination property of the algorithm or violate the failure detector's strong completeness property. To violate the termination

Goal 2: Violate liveness properties of the system

1. Cause liveness properties of consensus to fail
 - (a) Violate termination property (OR)
 - i. Attack availability
 - A. Attack network (OR)
 1. Physical destruction of network (OR)
 2. Denial of service attack on network
 - B. Attack process (OR)
 1. Physical destruction of processes (OR)
 2. Denial of service attack on processes
 - C. Attack smartcard
 1. Physical destruction of the smartcard (OR)
 2. Denial of service attack on the smartcard (OR)
 3. Pull smartcard out from smartcard reader
 - (b) Attack failure detector
 - i. Violate strong completeness property
 - A. Inject false messages (OR)
 - B. Inject replays of previous messages

Fig. 13. Attack tree for the threat of liveness properties.

property of consensus the attacker must try to avoid that the majority of correct processes eventually decide on some value by attacking the availability of either the network or the processes or the smartcard.

Attacks Aimed at Information Flow Properties of the System Figure 14 shows an attack tree for the threat of the system’s information flow properties. The goal is to violate the information flow properties of the system. For this purpose, attackers can attack the system’s confidentiality or do traffic flow analysis. To attack confidentiality, they either can try to read the encrypted messages transmitted over the network or can attack their own smartcard. There are many ways to read encrypted messages. Here, we consider only some of the most common ones. Note that in the majority of cryptographic systems, the secrecy of the method to encrypt data is based on the encryption algorithm, which is the collection of the mathematical rules determining the sequence of fulfilling the elementary operations above the data, and on the cryptographic key, which determines the precise computation of the plaintext in ciphertext and vice versa.

6.2 Summary

We evaluated the security of the entire system and showed how security threats are countered by security enforcing functions and mechanisms. To model the security threats against the system we made use of attack trees which provide a formal, methodical way of describing the security of systems, based on varying attacks. We examined three attacker goals and created an attack tree for each of these goals. The main goals of an attacker are to violate the safety, the liveness, and the information flow properties of the system. We showed the success or failure of the leaf nodes by analyzing the attack each leaf node presented in detail and identified the conditions for the attack as well as the capabilities needed by the attacker.

The analysis indicates that the system is secure against almost all discussed attacks. An attacker can only be successful in violating the safety property of the system by attacking the eventual strong accuracy property of the failure detector. However, the attacker is not successful in preventing the execution of the consensus algorithm. Weak physical protection of system components could make the system vulnerable to attacks but in order to be efficient a large amount of system components must be attacked what makes this type of attack highly unlikely. Attention should also be paid to side-channel attacks against smartcards since all cryptographic algorithms are assumed to be vulnerable to side-channel cryptanalysis if there are no special countermeasures in the implementation. Due to the large complexity and effort to perform a side-channel attack makes this type of attack also highly unlikely.

Goal 3: Violate information flow properties of the system

1. Attack Confidentiality (OR)
 - (a) Read encrypted message in transfer (OR)
 - i. Decrypt the message itself (OR)
 - A. Mathematically break asymmetric encryption (OR)
 - B. Ciphertext-only attack
 - ii. Obtain private key of recipient (OR)
 - A. Brute-force attack (OR)
 - B. Mathematically break asymmetric encryption (OR)
 - C. Social engineering (OR)
 - D. Ciphertext-only attack (OR)
 - E. Known-plaintext attack
 - iii. Get recipient to (help) decrypt the message
 - A. Chosen-plaintext attack (OR)
 - B. Adaptive chosen-plaintext attack (OR)
 - C. Chosen-ciphertext attack (OR)
 - D. Adaptive chosen-ciphertext attack (OR)
 - E. Read message after it is decrypted by the recipient
 - (b) Attack smartcard
 - i. Side-channel attack (OR)
 - ii. Physical attack
2. Traffic flow analysis
 - (a) Analyze traffic to/from own security module (OR)
 - (b) Analyze network traffic
 - i. Install sniffer (OR)
 - ii. Man-in-the-middle attack

Fig. 14. Attack tree for the threat of information flow properties.

7 Conclusions and Future Work

Here we presented a modular redesign of TrustedPals, a smartcard-based security framework capable of efficiently solving secure multiparty computation (SMC) by reducing it to fault-tolerant consensus between smartcards. The modular redesign allows TrustedPals to have consensus solutions which make use of failure detection, or more precisely, of the eventually perfect failure detector.

There are multiple lines of future work to consider. On the practical side as next step we intend to have the approach implemented and extended to other classes of failure detectors. On the theoretical side it would be interesting to study the minimal storage and communication effort necessary to solve consensus, since we use unbounded buffers in our implementation and the bit complexity of the messages we use is rather high. Also it is necessary to investigate the timing assumptions still further since in theory smart cards can also be slowed down arbitrarily. In such cases the assumptions of partial synchrony may not hold and we come close to a truly asynchronous system where consensus cannot be solved [9]. Investigating realistic models of smartcard-based systems that reflect this type of attack and still allow TrustedPals to be implemented will further broaden the applicability of the framework in practice.

In the trusted system however, security modules may not have their own source of activation (and therefore no realtime clock), so an untrusted host *can* on the one hand slow down its smartcard arbitrarily by disturbing the clock speed. On the other hand, the host *cannot* speed up the clock of the security module arbitrarily. This still results in a model of partial synchrony for the trusted system as we now explain.

Consider for example the situation depicted in Fig. 15. There a channel directed from a security module q to a security module p is shown. The correct process q sends heartbeat messages in constant time intervals to the malicious process p . The usual strategy in this case is that the receiver p sets a timeout $\Delta_p(q)$ of p for q . If p does not receive a message from q within this timeout interval, p suspects q to have failed. Now, if p manipulates its clock so that its smartcard processes faster and as a result the timeout intervals $\Delta_p(q)$ are shortened, p will assume q to have failed until it receives a message from q . It might happen finitely often that p times-out on q . After each timeout on q $\Delta_p(q)$ grows, and eventually the bounds on process

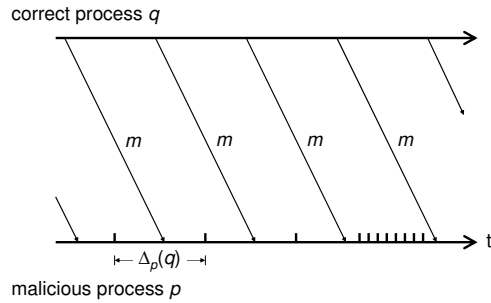


Fig. 15. Channel directed from a correct process q to a malicious process p .

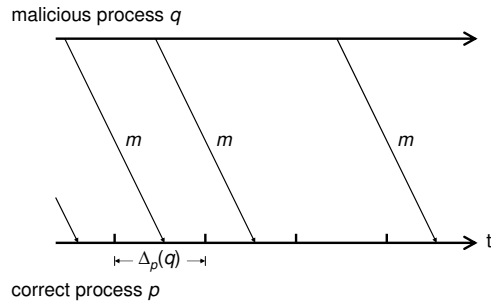


Fig. 16. Channel directed from a malicious process q to a correct process p .

speed and message delay hold. Thus, eventually process p cannot timeout on process q anymore, and the malicious host will have no impact on the system's timing assumption.

The inverse scenario is shown in Fig. 16. Here, the channel from a malicious host q to a correct host p is considered. Process q may intentionally withhold a message m and delay the sending of m for an arbitrarily amount of time. This will cause process p to assume that q has failed. If q finally sends m , p will not assume q to have failed anymore. This might also happen infinitely often. Assuming the system to be synchronous or partially synchronous and that there are no send omissions, the arbitrary delay caused by the malicious process will violate the timing assumption of the system.

References

1. G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolic. Gracefully degrading fair exchange with security modules. In *Proceedings of the Fifth European Dependable Computing Conference*, pages 55–71. Springer-Verlag, April 2005.
2. G. Avoine and S. Vaudenay. Optimal fair exchange with guardian angels. In *International Workshop on Information Security Applications (WISA)*, LNCS, volume 4, 2003.

3. Z. Benenson, M. Fort, F. Freiling, D. Kesdogan, and L. D. Penso. Trustedpals: Secure multiparty computation implemented with smartcards. In *11th European Symposium on Research in Computer Security (ESORICS)*, pages 306–314. Springer-Verlag, September 2006.
4. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
5. Z. Chen. *Java Card Technology for Smart Cards - 1st Edition*. Addison-Wesley Professional, 2000.
6. M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS-Real-time distributed security kernel. In F. Grandoni and P. Thévenod-Fosse, editors, *Dependable Computing - EDCC-4, 4th European Dependable Computing Conference, Toulouse, France, October 23-25, 2002, Proceedings*, volume 2485 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2002.
7. C. Delporte-Gallet, H. Fauconnier, and F. C. Freiling. Revisiting failure detection and consensus in omission failure environments. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC05)*, Hanoi, Vietnam, Oct. 2005.
8. C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
9. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
10. F. C. Freiling, R. Guerraoui, , and P. Kouznetsov. The failure detector abstraction. Technical report, Department for Mathematics and Computer Science, University of Mannheim, 2006.
11. V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.
12. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
13. P. MacKenzie, A. Oprea, and M. Reiter. Automatic generation of two-party computations. In *SIGSAC: 10th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2003.
14. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — A secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, Aug. 2004.
15. K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.
16. P. Sousa, N. F. Neves, and P. Verssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 686–690. Springer-Verlag, April 2006.
17. A. C. Yao. Protocols for secure computations. In *Proceedings of the Twenty-Third Annual Symposium on Foundations of Computer Science*, pages 160–164. Springer-Verlag, November 1982.