Secure Failure Detection in TrustedPals*

Roberto Cortiñas[1], Felix C. Freiling[2], Marjan Ghajar-Azadanlou[3], Alberto Lafuente[1], Mikel Larrea[1], Lucia Draque Penso[2], and Iratxe Soraluze[1]

[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. This paper presents 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. Within the redesign we investigate the problem of solving consensus in a general 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.

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) [22]. 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.

* Work by the Spanish authors was supported by the Spanish Research Council, under grant HA2005-0078. Work by the German authors was supported by DAAD PPP Programme *Acciones Integradas Hispano Alemanas*.

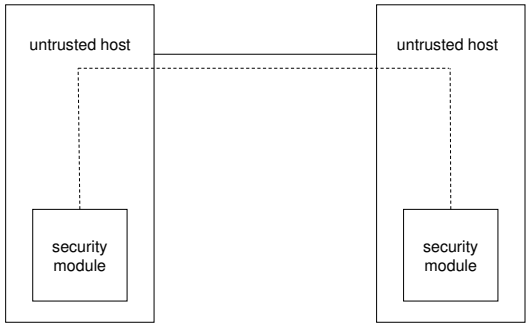

Fig. 1. Processes with tamper proof security modules.

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 [13]. 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).

Related Work. Delporte, Fauconnier and Freiling [8] 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 [8] 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 [8] implement the less general class Ω. Furthermore, Delporte, Fauconnier and Freiling [8] 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.

Apart from Delporte, Fauconnier and Freiling [8], other authors also investigated solving consensus in systems with omission faults. Unpublished work by Dolev et al. [10, 9] also follows the failure detector approach to solve consensus, however they focus on the class $\diamond\mathcal{S}(om)$ of failure detectors. Babaoglu, Davoli and Montresor [19] also follow the path of $\diamond\mathcal{S}$ to solve consensus in partitionable systems.

Recently, solving SMC *without* security modules has received some attention focusing on two-party protocols [17, 18]. 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 [21]. 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.

Our work on TrustedPals can also be regarded as building failure detectors for arbitrary (*Byzantine*) failures which has been investigated previously (see for example Kihlstrom, Moser and Melliar-Smith [15] and Doudou, Garbinato and Guerraoui [11]). In contrast to previous work on Byzantine failure detectors, we use security modules to avoid the tar pits of this area.

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 [12].
- 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.

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. For lack of space, the correctness proofs of the algorithms as well as more details on the security evaluation can be found elsewhere [7].

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.

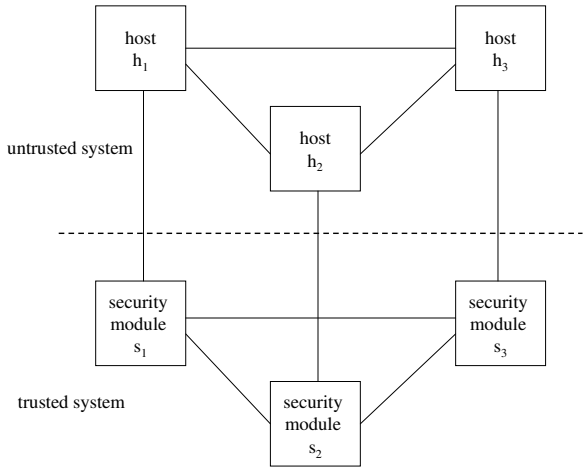

Fig. 2. The untrusted and trusted system.

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 [12]. 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* [16]. 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 [14] and later generalized by Perry and Toueg [20]. 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 no omissions from every out-connected process to every correct process, and hence to every in-connected process.

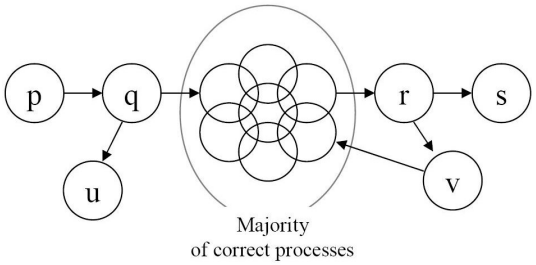

Fig. 3. Examples for classes of processes.

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.

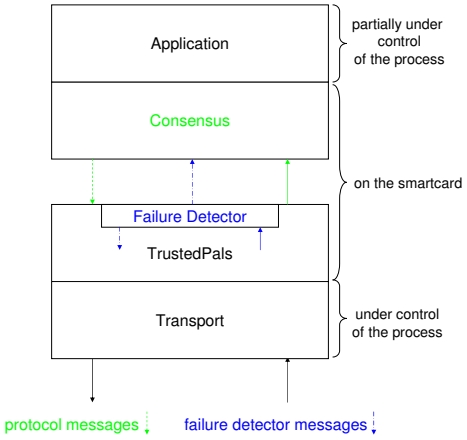

Fig. 4. The architecture of our system.

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.

Figs. 5, 6 and 7 present 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.

In order to detect message omissions, messages carry a sequence number. 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 element in the matrix has a value of 1. 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 q to p. We can derive from powers of the adjacency matrix if there is a path with no omission of any length between every pair of processes. Observe that 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 lists of in-connected and out-connected processes are computed in the *update_In_Out_Connected_lists*() procedure, which is called every time a value of the matrix M_p is changed.

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

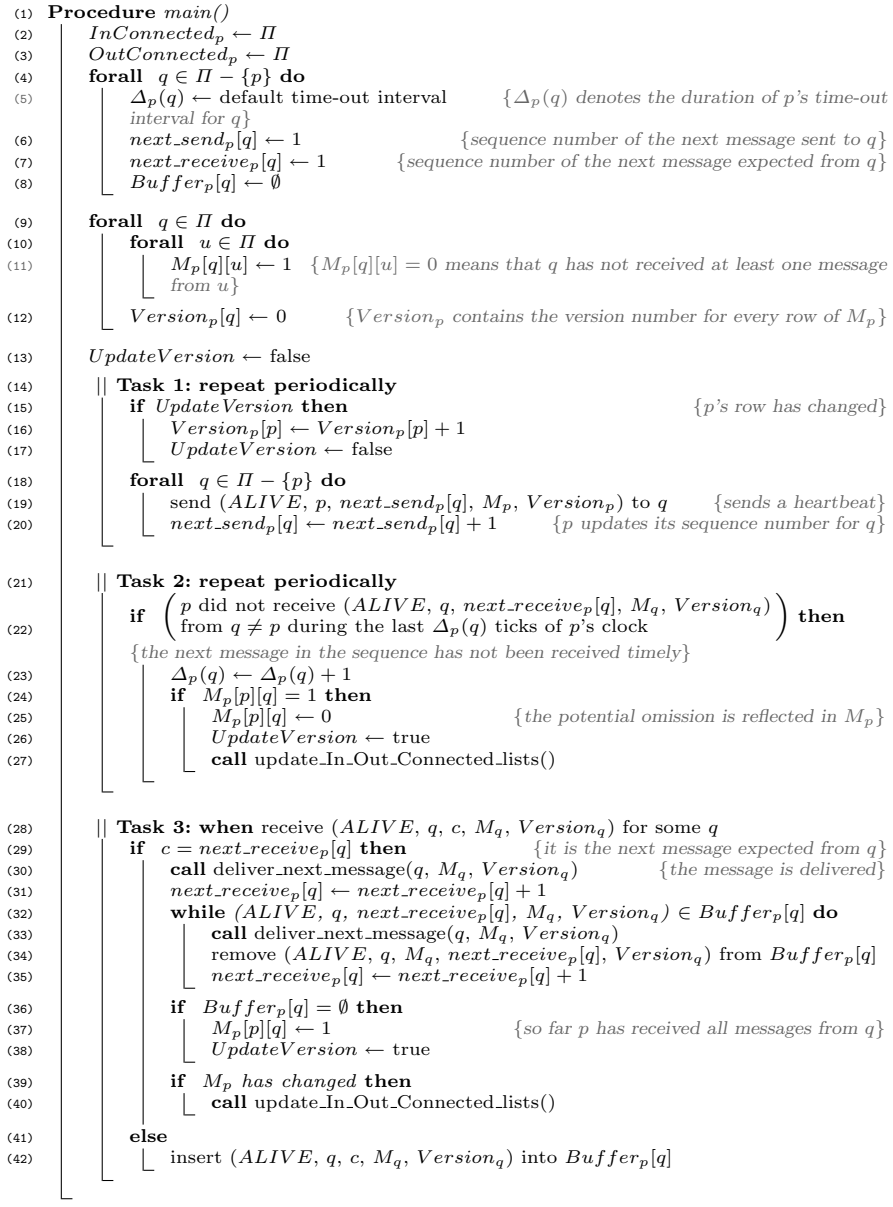

Fig. 5. $\diamond\mathcal{P}$ in the omission model: main algorithm.

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

```
Result: InConnected_p and OutConnected_p lists
(43) Procedure update_In_Out_Connected_lists()
(44)     A_p ← (M_p)^n                                   {A_p is the n-th power of the M_p matrix}
(45)     forall u, v ∈ Π do
(46)         if A_p[u][v] > 0 then
(47)             A_p[u][v] ← 1
(48)     In ← ∅
(49)     Out ← ∅
(50)     forall q ∈ Π do
(51)         if (Σ_{i=0}^{n-1} A_p[q][i] ≥ ⌈(n+1)/2⌉) then
(52)             In ← In ∪ {q}
(53)         if (Σ_{i=0}^{n-1} A_p[i][q] ≥ ⌈(n+1)/2⌉) then
(54)             Out ← Out ∪ {q}
(55)     InConnected_p ← In
(56)     OutConnected_p ← 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 ∈ Π do                                 {q's row of M_q is systematically copied into M_p}
(59)         M_p[q][v] ← M_q[q][v]
(60)     forall u ∈ Π − {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 ∈ Π do
(63)                 M_p[u][v] ← M_q[u][v]
(64)             Version_p[u] ← Version_q[u]
```

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

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.

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.

Figs. 8 and 9 present 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.

The correctness proof of the algorithm can be found in [7].

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

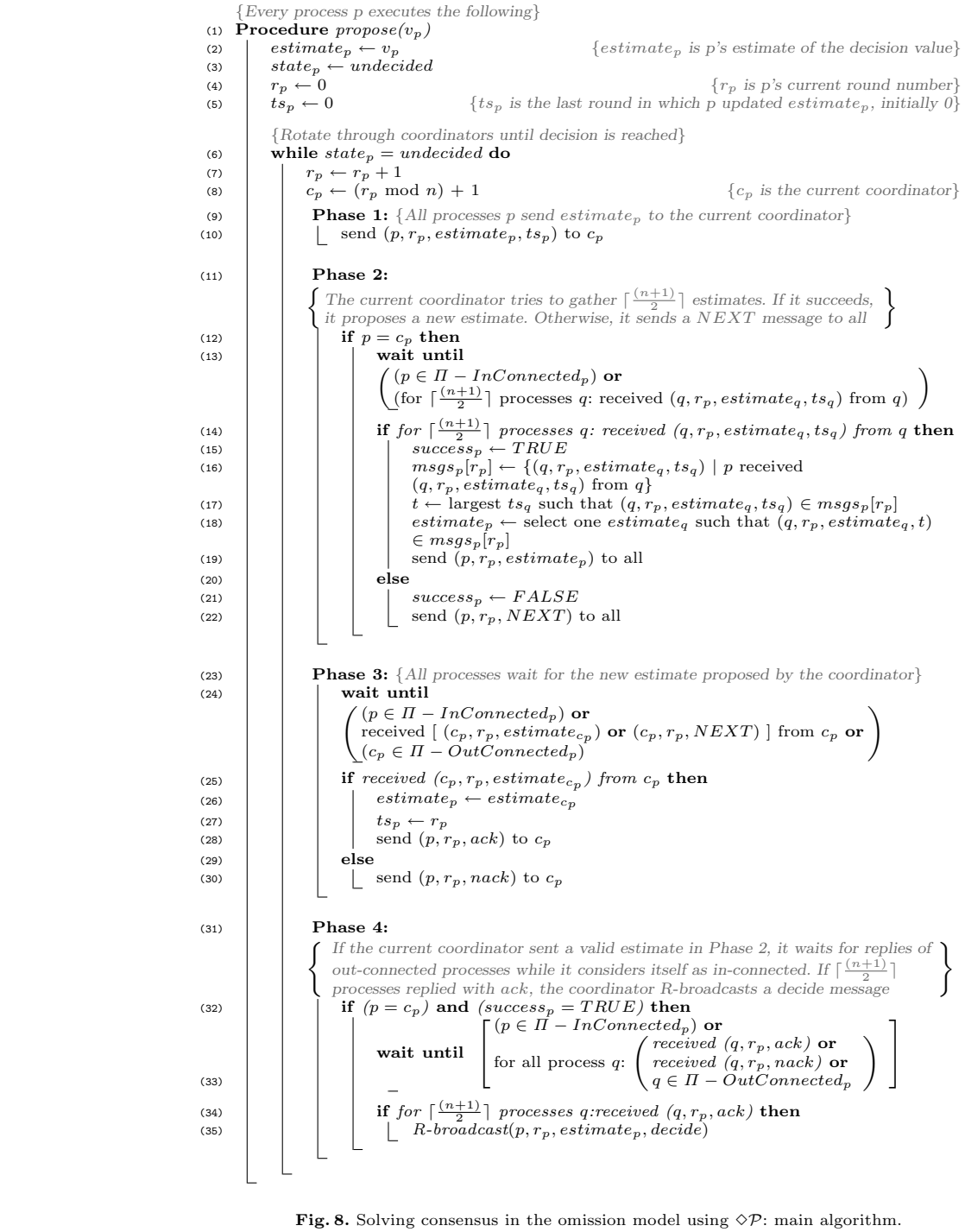

Fig. 8. Solving consensus in the omission model using $\diamond\mathcal{P}$: main algorithm.

```
    {If p R-delivers a decide message, p decides accordingly}
(36) when R-deliver(q, r_q, estimate_q, decide) do
(37)     if state_p = undecided then
(38)         decide(estimate_q)
(39)         state_p ← decided
```

Fig. 9. Solving consensus in the omission model using $\diamond\mathcal{P}$: adopting the decision.

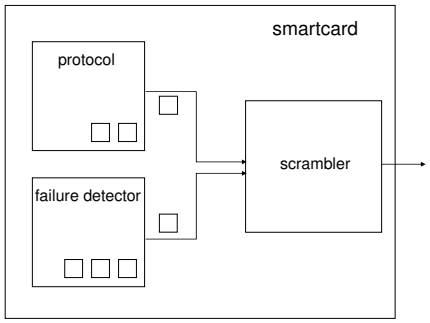

Fig. 10. Smartcard with scrambler.

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. 10). 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 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. 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.

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. R. Cortiñas, F. C. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L. D. Penso, and I. Soraluze. Secure Failure Detection in TrustedPals. Technical Report EHU-KAT-IK-07-07, The University of the Basque Country, July 2007. Available at `http://www.sc.ehu.es/acwlaalm/`.
8. 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.
9. D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Cornell University, Computer Science Department, Sept. 1996.
10. D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 286. Springer-Verlag, July 1997.
11. A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. In *Proceedings of the Int. Conference on Reliable Software Technologies*, Vienna, May 2002.
12. 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.
13. F. C. Freiling, R. Guerraoui, , and P. Kouznetsov. The failure detector abstraction. Technical report, Department for Mathematics and Computer Science, University of Mannheim, 2006.
14. V. Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. also published as Technical Report TR11-84.
15. K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1), 2003.
16. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
17. 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.
18. 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.
19. Özalp Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Softw. Eng.*, 27(4):308–336, 2001.
20. 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.
21. P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 686–690. Springer-Verlag, April 2006.
22. 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.