Secure Failure Detection and Consensus in TrustedPals

R. Cortiñas, F. C. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L. D. Penso, I. Soraluze

Abstract

We present a modular redesign of *TrustedPals*, a smartcard-based security framework for solving *Secure Multiparty Computation* (SMC). Originally, TrustedPals assumed a synchronous network setting and allowed to reduce SMC to the problem of *fault-tolerant consensus* among smartcards. We explore how to make TrustedPals applicable in environments with less synchrony and show how it can be used to solve *asynchronous* SMC. 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 $\Diamond\mathcal{P}$ of failure detectors in the omission model, which we call $\Diamond\mathcal{P}(om)$, and show how to implement $\Diamond\mathcal{P}(om)$ and have consensus in such a system with very weak synchrony assumptions, some of which new. The integration of failure detection and consensus into the TrustedPals framework uses tools from privacy enhancing techniques such as message padding and dummy traffic.

Index Terms

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

I. 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

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*.

A preliminary version of this paper appeared in the proceedings of the 9th International Symposium on Safety, Security and Stabilization (SSS 2007), Paris, France, 2007.

September 28, 2011

each other, nor the channels by which they communicate. This is the problem of *Secure Multiparty Computation* (SMC) [1]. 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 online auctions, sharing of signature or decryption functions and so on. Unfortunately, solving SMC is — without extra assumptions — very expensive in terms of communication (number of messages), resilience (amount of redundancy) and time (number of synchronous rounds).

TrustedPals [2] is a smartcard-based security framework for solving 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. In practice, processes are implemented as a Java desktop application and security modules are realized using Java Card Technology enabled smartcards [3], tamper-proof Subscriber Identity Modules (SIM) [4] like those used in mobile phones, or storage cards with builtin tamper-proof processing devices [5]. Roughly speaking, solving SMC among processes is achieved by having 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. Roughly speaking, within this secure overlay arbitrary and malicious behavior of an attacker is reduced to process crashes and message omissions. TrustedPals therefore allows to reduce the security problem of SMC to a problem of fault-tolerant synchronization [2], an area which has a long research tradition in fault-tolerant distributed computing (see for example Lynch [6]). However, solving the synchronization problem alone is not trivial, especially since we investigate it under message omission failures, a failure scenario which is rather unusual. Furthermore, the reduction from security to fault-tolerance creates a new set of validation obligations regarding the *integration* of a fault-tolerant algorithm into a secure system, which is also far from being trivial.

The initial definition of TrustedPals and its implementation assumed a *synchronous* network setting, i.e., a setting in which all important timing parameters of the network are bounded and known. 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 show how to solve the *asynchronous version* of SMC using asynchronous synchronization algorithms inspired by recent results in fault-tolerant distributed computing: we use an *asynchronous* consensus algorithm and encapsulate (some very weak) timing assumptions, including a new one, within a device known as a *failure detector* [7].

The concept of a failure detector has been investigated in quite some detail in systems with merely crash faults [8]. In such systems, correct processes (i.e., processes which do not crash) must eventually permanently suspect crashed processes. There is very little work on failure detection and consensus in message omission 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 does not crash but just omits some messages). In this work, instead of correct processes, we will consider *well-connected* processes, i.e., those processes which are able to compute and communicate without omissions with a majority of processes.

Related Work.

Our work on TrustedPals can be regarded as building failure detectors for arbitrary (*Byzantine*) failures, which has been investigated previously (see for example Malkhi and Reiter [9], Kihlstrom, Moser and Melliar-Smith [10], Doudou et al. [11], Doudou, Garbinato and Guerraoui [12], Haeberlen, Kouznetsov and Druschel [13], and more recently Haeberlen and Kuznetsov [14]). In contrast to previous works on Byzantine failure detectors, we use security modules to avoid the tar pits of this area. This contrasts TrustedPals to the large body of work that tackles Byzantine faults *directly*, like Castro and Liskov's "Practical Byzantine Fault-Tolerance" [15] or more recently the *Aardvark* system by Clement et al. [16] and the *Turquois* protocol by Moniz et al. [17]. While being conceptually simpler, Byzantine-tolerant protocols necessarily have to assume a two-thirds majority of correct processes in non-synchronous settings [18] while TrustedPals needs only a simple majority (due to the availability of the secure overlay network). Next to an improved resilience, TrustedPals by design can provide *secrecy* of data against attackers, a notion that can only be achieved in Byzantine-tolerant algorithms by applying complex secret sharing mechanisms [19]. All these advantages result from using security modules to constrain the attacker in such a way that

Byzantine faults are reduced to general omission faults.

Delporte, Fauconnier and Freiling [20] were the first to investigate non-synchronous settings in the TrustedPals context, always with the (implicit) motivation to make TrustedPals more practical. Following the approach of Chandra and Toueg [7] (and similar to this paper) they separate the trusted system into an asynchronous consensus layer and a partially synchronous failure detection layer. The main difference however is that they assume that transient omissions are masked by a piggybacking scheme while we detect transient omissions and, therefore, we do not need to piggyback all message history. Besides, they solve a *different version of consensus* than we do: roughly speaking, message omissions can cause processes to only be able to communicate *indirectly* and we admit processes to participate in consensus even if they cannot communicate directly. Delporte, Fauconnier and Freiling [20] only guarantee that all processes that can communicate *directly* with each other solve consensus. In contrast, we allow also another set of processes to propose and decide: those which are able to send and receive messages even indirectly. As a minor difference, we focus on the class $\Diamond\mathcal{P}$ of eventually perfect failure detectors whereas they [20] implement the Ω failure detector. Furthermore, they [20] do not describe how to integrate failure detection and consensus within the TrustedPals framework: a realistic adversary who is able to selectively influence the communication messages of the algorithms for failure detection and consensus can cause their consensus algorithm to fail. This problem is partly addressed in a recent paper [21] where consensus and failure detection are integrated for efficiency purposes, not for security.

Apart from Delporte, Fauconnier and Freiling [20], other authors also investigated solving consensus in systems with omission faults. Work by Dolev et al. [22], [23] 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 [24] also follow the path of $\Diamond\mathcal{S}$ to solve consensus in partitionable systems. Alternatively, Santoro and Widmayer [25] assume a synchronous system model, and Moniz et al. [26] use randomization.

Recently, solving SMC *without* security modules has received some attention focusing mainly on two-party protocols [27]–[32]. In systems *with* security modules, Avoine and Vaudenay [33] examined the approach of jointly simulating a TTP. This approach was later extended by Avoine et al. [34] 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. [2] 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.

Ben-Or et al. [35] were the first to investigate solutions to the *asynchronous* variant of SMC which is slightly more involved than its synchronous counterpart because the failure to provide input to F by some party cannot be attributed solely to that party — it can also be due to unpredictable delays in the network. This has consequences regarding the resilience of SMC. While it is possible to solve SMC in the synchronous setting with a simple majority of benign processes [36], [37], in asynchronous settings a two-third majority is necessary [38].

Correia et al. [39] 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 [40]. However, the adversary model of Correia et al. [39] 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 works [2], [33], [34], Correia et al. [39] also assume a synchronous network model at least in a part of the system.

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 show how to solve asynchronous Secure Multiparty Computation by implementing TrustedPals in asynchronous systems with a (weak) failure detector. We do this by reducing the problem of SMC to the problem of uniform consensus in omission failure environments. As a corollary we show that in systems with security modules and weak timing assumptions the resilience of asynchronous SMC can be improved from a two-thirds majority to a simple majority of benign processes.
- We propose a new definition of connectedness in omission environments. Informally, a process is *in-connected* if it does not crash and, despite omissions, receives either directly or indirectly all messages that a majority of processes sends to it. Similarly, a process is *out-connected* if it does not crash and all messages it sends are received by a majority of processes. We also consider *well-connected* processes, which are those processes that are both in-connected and out-connected.
- We give a novel definition of consensus in the new omission model, by refining the termination property of consensus (“Every in-connected process eventually decides some

value"), and an algorithm which uses the failure detector class $\Diamond\mathcal{P}(om)$ to solve consensus. That algorithm is an adaptation of the classic algorithm by Chandra and Toueg for the crash model.

- We give a novel definition of $\Diamond\mathcal{P}$ in the omission model, $\Diamond\mathcal{P}(om)$, and we show how to implement it in a system with weak synchrony assumptions in the spirit of partial synchrony.
- 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 Section II we give an overview over TrustedPals, its architecture and the motivation behind the definitions and model and show how asynchronous SMC can be reduced to uniform consensus. In Section III we fully formalize the system model of TrustedPals. In Section IV we show how to solve consensus using an abstract failure detector of the class $\Diamond\mathcal{P}(om)$. In Section V we show how to implement the failure detector $\Diamond\mathcal{P}(om)$ in the omission failure model under very weak synchrony assumptions. In Section VI we describe how to integrate failure detection and consensus securely in the TrustedPals framework. We conclude in Section VII.

II. TRUSTEDPALS IN WEAKLY SYNCHRONOUS SYSTEMS

We now give an overview over the TrustedPals system architecture and show how the reduction from SMC to consensus can be done in non-synchronous systems. Besides giving the reduction, this section is meant as an informal introduction giving a high level view of the definitions used in the remainder of the paper. All necessary notions are fully formalized later in Section III.

A. Untrusted and Trusted System

We formalize the system assumptions within a hybrid model, i.e., the model is divided into two parts (see Fig. 1). The lower part consists of n processes which represent the *untrusted hosts*. The upper part equally consists of n processes which represent the *security modules*. Because of the lack of mutual trust among untrusted hosts, we call the former part

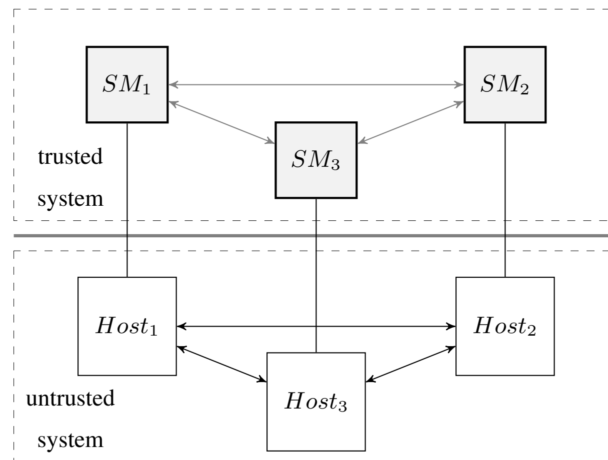

Fig. 1. The untrusted and trusted system.

the *untrusted system*. Since the security modules trust each other we call the latter part the *trusted system*.

The processes in the untrusted system (i.e., the hosts) execute (possibly untrustworthy) user applications like e-banking or e-voting programs. Because of the untrustworthy nature of these processes, they use the trusted system as a subsystem to solve the involved security problems. The trusted system consists of software running inside the security modules. This software must have been certified by some accepted authority. It is not possible for the user to install arbitrary software on the trusted system. The tamper-proof nature of the trusted processes allows to protect stored and transmitted information even from the untrusted processes on which they reside. The authority can be an independent service provider (like a network operator) and is only necessary within the *bootstrap* phase of the system, not during any *operational* phases (like running the SMC algorithms).

Formally, the connection between the untrusted and trusted system is achieved by associating each process in the untrusted system (i.e., each host) with exactly one process in the trusted system (i.e., a security module) and vice versa. Hence, every untrusted process has a "trusted pal" (an associated trusted process). Since host and security module reside on the same physical machine, we assume that for each association there is a bidirectional eventually timely and secure communication channel, e.g., implemented by shared variables or message passing communication in the host operating system.

For better readability, we sometimes refer to untrusted processes simply as *hosts* and to trusted processes simply as *security modules*. We refer to TrustedPals as the collection of

trusted processes together with the services they provide to the untrusted processes.

B. Defining Asynchronous SMC with TrustedPals

As mentioned in the introduction, SMC allows the set of hosts to correctly compute some common function F of their local values. While the original work on TrustedPals [2] used the synchronous definition of SMC given by Goldreich [41], we follow the definition of *asynchronous SMC* given by Ben-Or et al. [35]. The difference between the synchronous and asynchronous form of SMC lies in the fact that F is not computed on *all* inputs but rather on a subset of inputs of size at least $n - f$, where f is an upper bound on the number of faulty (i.e., non-benign) processes. A process is *benign* if, besides not crashing, it follows its protocol (i.e., it is non-Byzantine).

More formally, let $x_1, \ldots, x_n$ be the private inputs of each host. In asynchronous SMC, the result r is computed on the inputs from a subset C of hosts of size at least $n - f$, i.e., $r = F(y_1, \ldots, y_n)$ where $y_i = x_i$ if host i is in C and some default value (e.g., $y_i = 0$) otherwise. The result r should be computed reliably and securely, i.e., as if all hosts were using a trusted third party (TTP). This means that the individual inputs remain secret to other hosts (apart from what is given away by r) and that malicious hosts can neither prevent the computation from taking place nor influence r in favorable ways.

In this paper, we use the following definition of (asynchronous) SMC: A protocol solves *secure multi-party computation (SMC)* with TrustedPals if it satisfies the following properties [35], [41]:

- *SMC-Validity*. If a host receives a result, then that result was computed by applying F on the inputs from a subset C of hosts of size at least $n - f$, i.e., $r = F(y_1, \ldots, y_n)$ where $y_i = x_i$ if host i is in C and some default value (e.g., $y_i = 0$) otherwise. The set C is the same for all hosts that receive a result.
- *SMC-Agreement*. No two values returned by security modules differ.
- *SMC-Termination*. Every benign host eventually receives a result from its associated security module.
- *SMC-Privacy*. Faulty hosts learn nothing about the input values of benign hosts (apart from what is given away by the result r and the input values of all faulty hosts).

Recall that these properties abstractly specify what happens when a TTP is used to solve the problem [35], [41].

C. Untrusted System: Assumptions

Within the untrusted system each pair of hosts is connected by a pair of unidirectional communication channels, one in each direction. We assume that there is a minimal set of reliable channels in the system (this will be formalized in Section III). The rest of the channels can be lossy. Recall that every message inserted to a reliable channel is eventually delivered at the destination. We assume no particular ordering relation on channels.

Failure model: The process failure model we assume in the untrusted system is the *Byzantine failure model* [42]. A Byzantine process can behave arbitrarily. We will assume a majority of benign processes in the untrusted system.

Timing: We assume that a local real-time clock is available to each process in the untrusted system, but clocks are not necessarily synchronized within the network. The untrusted system is assumed to be *partially synchronous*, meaning that eventually unknown bounds on processing and communication delays hold for the majority of benign processes. The model is a variant of the partial synchrony models of Dwork, Lynch and Stockmeyer [43]. The differences are that the bounds must hold for just a majority of processes, and that we assume a set of reliable, eventually timely channels connecting those processes.

D. Trusted System: Assumptions

The trusted system can be considered as an *overlay network* — a network that is built on top of another network — over the untrusted system. Nodes in the overlay network can be thought of as being connected by virtual or logical channels. In practice, for example, smartcards could form the overlay network which runs on top of the Internet modeled by the untrusted processes. In the trusted system each process has also an outgoing and an incoming communication channel with every other process.

Trust Model: Within the trusted system we assume that any two communicating parties can establish mutual message confidentiality, message integrity and message authentication. This can be realized, for example, by exchanging cryptographic keys during a setup phase of the system. As mentioned above, we assume that the code running within the trusted system has been certified by some trusted authority, i.e., nodes in the trusted system may assume that each other's programs have not been tampered with. The trusted authority acts only during the setup phase of the system, not during the operational phase.

Timing: Security modules do not have any clock, they just have a simple step counter, whereby a step consists of possibly consuming a message, executing a local computation

and possibly sending a message. Passing of time is checked by counting the number of steps executed. Roughly speaking, the timing assumptions for the processes in the trusted system are the same as those of the untrusted system, i.e., we assume partial synchrony. However, as we will explain next, in case the trusted process is associated with an untrusted process which is faulty/malicious, the trusted process may not rely on any timing assumptions whatsoever.

Failure Model: Like the untrusted system, the trusted system is also prone to attacks. However, the assumptions on the security modules and the possibility to establish secure channels reduce the options of the malicious hosts to attacks on the liveness of the system, i.e., (1) destruction of the security module, (2) interception of messages between the channel and the security module or (3) changes in the frequency of the step counter. This way, in the trusted system we assume the failure model of *general omission* and some asynchrony that can only affect those trusted processes associated with faulty processes in the untrusted system (i.e., security modules residing on Byzantine hosts).

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 [44] and later generalized by Perry and Toueg [45]. In the *general omission* model processes can fail by crashing or experience either *send omissions* or *receive omissions*. In our system we allow the possibility of *transient* omissions, i.e., a process may temporarily drop messages and later on reliably deliver messages again. Of course, *permanent* omissions are possible too.

Besides omissions, trusted processes can become arbitrarily slow (asynchronous) although the physical system in which they operate (i.e., the untrusted system) is partially synchronous. This models the effect of two different types of attacks by malicious hosts which we now explain: *timing attacks* and *buffering attacks*:

- Recall that security modules use a step counter to be aware of passing of time. The speed of the step counter is controlled by the associated host. In a *timing attack*, a malicious host arbitrarily changes the speed of the step counter of its security module. In that way, it can make the security module work faster (although the speed is physically bounded by the host's own clock) or slower (not bounded). As a consequence, the behavior of its security module could become asynchronous and the communication through all the virtual channels which are adjacent to that security module would become asynchronous as well.
- As we have previously pointed out, a malicious host can intercept messages between

its security module and the communication channels. Removal of intercepted messages is modeled as a message omission. In a *buffering attack*, the host does not remove the messages but stores them in a buffer and later on injects them into the communication channel after an arbitrary delay. This means that the message is not omitted but communication through that channel may become asynchronous in the trusted system.

Observe that both attacks affect the timing behavior of the attacked security module. However, buffering attacks are more selective, since a particular communication channel of a security module could be attacked (become asynchronous in the trusted system) without affecting the rest of the communication channels of that security module.

In addition to the previous types of attacks, message reordering attacks are treated as a particular case of message buffering attacks, since in our algorithms no message is delivered if it is not the expected one (messages carry a unique sequence number). Also, note that the case of buffer overflow in the smartcard (e.g., if the attacker buffers a lot of messages and then passes all those messages at the same time to the smartcard) can be naturally treated as if the smartcard omitted the reception of some message(s).

Classes of Processes: Earlier work on failure detectors and partial synchrony [7], [43] assumed a majority of correct processes in order to solve consensus. However, as observed earlier we allow *faulty* processes to participate in consensus provided that they keep their ability to compute (no crash) and to communicate without omissions with a majority of processes.

In order to circumvent some transient omissions, we admit the possibility of indirect communication between two processes. For example, if there are omissions in the communication channel from a process p to another process q, but both of them have no omissions with a third process r, process p could indirectly communicate with q through r without any omission. This way, a process will be considered a *well-connected* process as long as it is able to communicate with a majority of processes without any omission, even if it has suffered some omissions. The set of well-connected processes will be formalized in Section III.

Furthermore, we should notice that the *connectedness* of a process can be asymmetric, since it can suffer send omissions and receive omissions independently, e.g., a process can be able to send to a majority of processes, but not be able to receive from a majority of processes (because it has had too many receive omissions). Following this motivation, we consider the following classes of processes, based on their ability to communicate (we give only rough and intuitive definitions here and fully formalize these notions later in Section III):

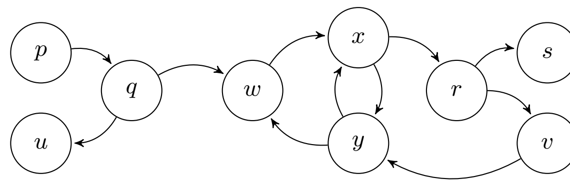

Fig. 2. Examples for classes of processes.

- A process is *in-connected* if it does not crash and it receives all messages that some well-connected process sends to it.
- A process is *out-connected* if it does not crash and all messages it sends to some well-connected process are received by that process.

Based on these definitions, well-connected processes are both in-connected and out-connected. Observe that every out-connected process can send information to any in-connected process with no omissions. Fig. 2 shows an example where arcs represent channels with no omissions. The majority of well-connected processes corresponds to the set $\{x, y, w, r, v\}$. Processes p and q are out-connected, while process s is in-connected. Finally, process u is neither in-connected nor out-connected.

Connecting the Failure Assumptions: To connect the failure assumptions from trusted and untrusted systems we make the following assumption: a benign process in the untrusted system, i.e., a benign host, implies that its associated process in the trusted system is well-connected. Since there is a majority of benign hosts, the previous assumption implies that there is a majority of well-connected processes in the trusted system. However, a non-benign host does not necessarily imply a non well-connected process in the trusted system.

E. Uniform Consensus

We are now ready to give the definition of consensus used in this paper. Intuitively, 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. The difference between (regular) consensus and *uniform* consensus lies in the uniform agreement property that demands that non-correct processes (if they decide) are not allowed to decide differently from correct processes.

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 experience some omissions, in-connected processes are those that will be able to receive the decision. The properties of uniform consensus in the omission model are thus 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.

F. Solving Asynchronous SMC with TrustedPals

The original work on TrustedPals [2] reduced the problem of SMC to that of *Uniform Interactive Consistency* (UIC) in the trusted system. Roughly speaking, within the trusted system, trusted processes exchange the inputs, then compute the function F and synchronize before returning the result back into the untrusted system. The implementation was based on an algorithm for uniform consensus by Parvédy and Raynal [46]. We now argue that the asynchronous variant of SMC can also be solved using *uniform consensus* in TrustedPals. We give pseudocode in form of a procedure in Fig. 3.

```
(1) Procedure SMC(x_i)
(2)   send (i, x_i) to all other trusted processes
(3)   wait until (for ⌈(n+1)/2⌉ processes q: received (j, x_j) from q)
(4)   V_i ← {all (j, x_j) received}
(5)   V ← uniform_consensus(V_i)
(6)   forall j ∈ {1, ..., n} do
(7)     if (j, x_j) ∈ V then y_j ← x_j else y_j ← 0   { some default value }
(8)   r ← F(y_1, ..., y_n)
(9)   return r
```

Fig. 3. Solving SMC using uniform consensus.

For every trusted process i, after receiving the input value x_i from its host, the trusted process sends x_i to all other processes in the trusted system. Thereafter it waits for the receipt of at least $\lceil (n+1)/2 \rceil$ values from other trusted processes and collects the pairs (j, x_j) in a local set V_i, where x_j is the value received from process j. Next, the set V_i is proposed to uniform consensus. Let V denote the decision value of uniform consensus. From V, the process constructs the vector of inputs for F (as used in the definition of SMC), computes F on that vector and returns the result to its host.

We now argue that this algorithm implements SMC for a majority of benign hosts if uniform consensus can be solved in the trusted system. Recall that a majority of benign hosts implies that a majority of processes in the trusted system will be well-connected and thus in-connected.

First consider SMC-Termination. Although the system is asynchronous, the distribution of input values to all other processes will terminate since only a majority of values has to be received and a majority of hosts are benign. Therefore, all the trusted processes of benign hosts will eventually propose a value to uniform consensus. From the Termination property of uniform consensus, every such process will eventually decide and return the computed value to its host.

Now consider SMC-Validity. Since all security modules on benign hosts enter uniform consensus, they all propose a set V_i containing a majority of values. From the Validity property of uniform consensus, the decided value V on every such process will also contain a majority of values. The uniform agreement property of uniform consensus in turn guarantees that any result returned will be computed on the same vector of inputs. SMC-Agreement trivially follows from the protocol and the uniform agreement property of uniform consensus.

Proving SMC-Privacy is much more intricate because it depends critically on how the trusted system (i.e., TrustedPals) operates. Intuitively, the trusted system should leak no information other than what is communicated at its interface (i.e., the input of x_i and the output of the result r). This will be subject of Section VI where we integrate all algorithms in the trusted system such that we achieve a form of *unobservability* [47], [48], a notion known from the area of privacy-enhancing techniques that theoretically closes all side channels through which confidential information may leak.

Fig. 4 summarizes the layers and interfaces of the proposed modular architecture for TrustedPals. A message exchange is performed in the transport layer, which is under control of the untrusted host. The security mechanisms for message encryption/decryption run in the layer termed “Security” on the security module. In the failure detector and consensus/SMC layers run the failure detection and consensus/SMC algorithms respectively. Finally, in the application layer, which again is under the control of the untrusted host, application software offering user interfaces to consensus/SMC operate.

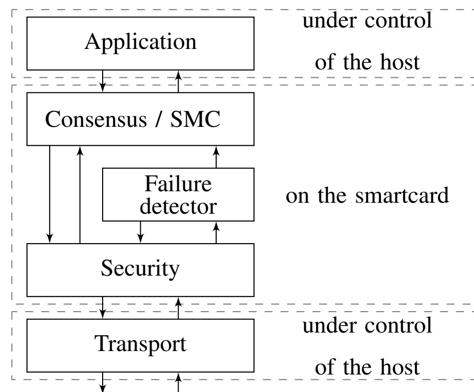

Fig. 4. The architecture of our system.

III. Formal System Model

Processes and Channels

We model a distributed system as a set of $n > 1$ processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ which are connected through pairwise bidirectional communication channels in a fully connected topology. In the following, we will also use p, q, r, etc. to denote processes. We denote the channel from p to q by c_{pq}.

Algorithms and Events

An algorithm A consists of a set of deterministic automata, one for each process. We give our algorithms in an event-based notation and thus assume that a local FIFO event queue is part of the local state of every process. Within an execution step, a process takes an event from the queue, performs a state transition according to the event, and then may send a message or add a new event to the queue. Message arrivals are treated as events too, i.e., when a message arrives, an appropriate event is added to the queue. It is "received" by the process when this event is processed. We assume that every process which does not crash executes infinitely many events.

Global Clock

We use a discrete global clock to simplify the presentation of our model. However, no process has access to this clock, it is merely a fictional device. For simplicity we take the range T of the clock to be the set of natural numbers.

Steps (i.e., event executions) on processes are always associated with a certain global time. We assume a linear model of event execution, i.e., for every instance in time there is at most one event in the system which is executed.

Process Failures

Processes can experience different kinds of failures: crash failures, omission failures and timing failures.

A *crash failure set* $F_c \subset \Pi$ is a subset of processes. Informally, F_c contains all processes that will eventually crash.

A *send-omission failure set* $F_{so} \subset \Pi \times \Pi$ is a relation over Π. Informally, $(p, q) \in F_{so}$ means that process p experiences at least one send omission towards q. If $(p, q) \notin F_{so}$ then p never experiences a send omission towards q.

Similarly, a *receive-omission failure set* $F_{ro} \subset \Pi \times \Pi$ is a relation over Π. Informally, $(p, q) \in F_{ro}$ means that process q experiences at least one receive omission from p. So if $(p, q) \notin F_{ro}$ then q never experiences a receive omission from p.

Some processes may experience timing failures. Timing failures refer to process asynchrony. We define an *asynchronous process failure set* $F_{ap} \subset \Pi$ as a subset of processes. Intuitively, F_{ap} contains all processes which are asynchronous in the system. Processes which are not in F_{ap} are eventually synchronous [43] meaning that their processing speed is eventually bounded. Formally, a process p is *synchronous* if there exists a known bound Ψ such that the time between the execution of any two steps of p is bounded by Ψ. A process p is *eventually synchronous* if there exists a time after which p is synchronous (additionally, Ψ can be unknown). Note that this implies that the relative process speeds between any pair of eventually synchronous processes is bounded. In our system model, both Ψ and the time after which Ψ holds are unknown.

A *process failure set* $F = (F_c, F_{so}, F_{ro}, F_{ap})$ is a tuple consisting of a crash failure set, a send-omission failure set, a receive-omission failure set, and an asynchronous process failure set.

We define the set of *correct processes* to be the set of all processes that neither crash nor experience any omission nor are asynchronous. We denote this set with $\mathcal{C}$. Formally:

$$\mathcal{C} = \{p \in \Pi : p \notin F_c \land p \notin F_{ap} \land (\forall q \in \Pi : (p, q) \notin F_{so} \land (q, p) \notin F_{ro})\}$$

As we will see, we do not need to assume the existence of a majority of correct processes. Moreover, the set of correct processes could even be empty.

Send and Receive

Processes can send a message using the *Send* primitive. The event *Send*(p, m, q, t) means that at time t process p sends m to q. In effect, m is inserted into the channel c_{pq} unless p experiences a send omission of m towards q. If c_{pq} is a reliable channel, for any message m inserted into c_{pq}, the channel guarantees that eventually an appropriate event is added to the local event queue of process q. However, due to buffering attacks or channel asynchrony/loss, there is not a time bound for the event in q to be added. In case c_{pq} is not reliable or q experiences a receive omission, m is removed from the channel without adding an appropriate event to the event queue of q. When this event is processed, we say that the message is received at time t, formalized as the occurrence of the event *Receive*(p, m, q, t). We allow processes to selectively wait for messages.

No particular order relations are defined in the reception of messages. We assume that every message m from p to q is tagged with a unique *sequence number* assigned by p.

Channel Failures

We say that a message m from p to q is *received timely* if the receive event of m happens at most Φ clock ticks after the send event of m, being Φ a time bound. We say that the channel c_{pq} is *eventually timely* if there exists a point in time t such that all messages from p to q sent and received after t are received timely. Formally:

$$\exists t \in T : \exists \textit{ bound } \Phi : \forall m : \textit{Receive}(p, m, q, t_r) \wedge \textit{Send}(p, m, q, t_s) \wedge t < t_s < t_r \Rightarrow (t_r - t_s \leq \Phi)$$

Again, in our partially synchronous system model both Φ and the time after which Φ holds are unknown.

We define an *asynchronous/lossy channel failure set* $F_{ac} \subset \Pi \times \Pi$ as a relation over Π such that $c_{pq} \in F_{ac}$ iff the channel c_{pq} is asynchronous and/or lossy. Note that if $c_{pq} \notin F_{ac}$ then c_{pq} is eventually timely and reliable. Note also that $c_{pq} \notin F_{ac}$ does not necessarily imply that $c_{qp} \notin F_{ac}$.

Considering both process and channel failures, we define now the relation $\rightarrow$ to denote an *eventually timely and failure-free direct communication* from p to q:

$$p \rightarrow q \Leftrightarrow (p \notin F_c \wedge p \notin F_{ap}) \wedge (q \notin F_c \wedge q \notin F_{ap}) \wedge (p, q) \notin F_{so} \wedge (p, q) \notin F_{ro} \wedge c_{pq} \notin F_{ac}$$

Note that not necessarily $p, q \in \mathcal{C}$. The relation $\rightarrow$ is reflexive, not symmetric and not transitive.

Indirect Communication

Two processes p and q may communicate directly through the channel c_{pq}, or indirectly using message relaying through some path of any length. For example, consider the following relaying mechanism that enables indirect communication:

- To send a message m_{pq} at time t_s from p to q, p executes:

$$\forall r \in \Pi - \{p\} : \textit{Send}(p, m_{pq}, r, t_s)$$

- The first time p executes the event $\textit{Receive}(-, m_{rs}, p, t_r)$, if $p \neq s$ then p executes:

$$\forall u \in \Pi - \{p, r\} : \textit{Send}(p, m_{rs}, u, t_r + \delta)$$

  where δ denotes the local processing time since the execution of the *Receive* event until the execution of the *Send* event.

We will address in Section VI how to achieve indirect communication without an explicit relaying mechanism.

Eventually Timely Communication

When considering indirect communication, messages from p can be eventually received timely by q despite the fact that the channel c_{pq} is not eventually timely and reliable. We first define the relation $\stackrel{*}{\rightarrow}$ to denote an eventually timely and failure-free (direct or indirect) communication from p to q. Informally, $p \stackrel{*}{\rightarrow} q$ means that there is a path from p to q such that for every adjacent processes r and s along the path, $r \rightarrow s$ is satisfied. Formally:

$$p \stackrel{*}{\rightarrow} q \Leftrightarrow (p \rightarrow q) \vee (\exists r : (p \rightarrow r \wedge r \stackrel{*}{\rightarrow} q))$$

The relation $\stackrel{*}{\rightarrow}$ is reflexive, not symmetric and transitive.

Observe that the relation $\stackrel{*}{\rightarrow}$ implies a *stable* eventually timely path from p to q. Observe also that the relation $\stackrel{*}{\rightarrow}$ does not impose any condition on the failure of the channel c_{pq}.

Nevertheless, a stable path is not actually required in order q to receive messages from p with a reliable and timeliness behavior which is indistinguishable from $p \stackrel{*}{\rightarrow} q$, as we now explain.

Let A be an algorithm that implements indirect communication as defined in the previous section. Assume that there is an adversary that knows A and that can manipulate the clock and/or retain messages in the buffers of the untrusted system, a feature of our system model. The adversary could for example delay for an unbounded time a message m_{pq} in a channel

c_{rs} between processes r and s in a path Z_{pq} from p to q. However, if m_{pq} is able to reach q through an alternative path Y_{pq}, including only channels such that the message is received timely by each process along the path, then m_{pq} will be delivered timely by q. If later the adversary delays a new message m'_{pq} in a channel c_{uv} of the path Y_{pq} from p to q then m'_{pq} could still be delivered timely by q through another path W_{pq} including only channels such that the message is received timely by each process along the path, and, interestingly, one of these channels could be c_{rs}. By playing this game forever in a run R of algorithm A, the adversary could prevent any *stable* eventually timely path from p to q. Observe also that the adversary could even manipulate the step counter based local clock in order to avoid time-outs triggering. However, if the behavior of the communication from p to q in R is *fair enough*, R will be indistinguishable from another run R' in which $p \xrightarrow{*} q$.

In order to formalize this kind of communication, we introduce a predicate β to denote that a message m sent through a channel c_{pq} is received timely by q, i.e., before q's local clock based current timeout $\Delta_q(p)$ expires. Formally, $\beta(m, c_{pq}) = \text{TRUE}$ iff an event *Receive*(q, m, p, t_r) occurs at q before $\Delta_q(p)$ expires.

We define the relation $\rightsquigarrow$ to denote a failure-free (direct or indirect) communication from p to q such that eventually the communication from p to q *holds local clock timeouts*.

$$p \rightsquigarrow q \Leftrightarrow p \notin F_c \wedge q \notin F_c \wedge p \notin F_{ap} \wedge q \notin F_{ap} \wedge (\forall m_{pq} : \forall t_s \in T : \textit{Send}(p, m_{pq}, q, t_s)$$
$$\Rightarrow (\exists t_r \in T : t_r > t_s : \textit{Receive}(-, m_{pq}, q, t_r)) \wedge (\exists t : t_s > t \wedge \exists Z_{pq} : \forall c_{uv} \in Z_{pq} : \beta(m_{pq}, c_{uv}) = \text{TRUE}))$$

Observe that, although delays may increase infinitely often, messages can be delivered without timing out on them. In a sense, this form of synchrony is more similar to the *progress assumptions* of the timed-asynchronous system model by Cristian and Fetzer [49].

The following relations hold:

$$p \xrightarrow{*} q \Rightarrow p \rightsquigarrow q$$
$$p \xrightarrow{*} q \wedge q \rightsquigarrow r \Rightarrow p \rightsquigarrow r$$
$$p \rightsquigarrow q \wedge q \xrightarrow{*} r \Rightarrow p \rightsquigarrow r$$

The relation $\rightsquigarrow$ is reflexive, not symmetric and transitive. Again, the relation $p \rightsquigarrow q$ does not impose any condition on the failure of the channel c_{pq}.

Well-Connected Processes

Two processes p, q belong to the same connected component if $p \xrightarrow{*} q$ and $q \xrightarrow{*} p$. Based on this, we define the set of well-connected processes $\overset{*}{\mathcal{C}}$ as the connected component formed by a majority of the processes in the system, i.e., $|\overset{*}{\mathcal{C}}| \geq \lceil \frac{(n+1)}{2} \rceil$.

Roughly speaking, every pair of processes in the set $\overset{*}{\mathcal{C}}$ can communicate reliably, eventually timely and without omissions between them. Recall that, since we assumed a majority of benign hosts (as presented at end of Section II-D), there is also a majority of well-connected processes in the trusted system, which guarantees the existence of $\overset{*}{\mathcal{C}}$. Observe also that every correct process is well-connected, i.e., $\mathcal{C} \subseteq \overset{*}{\mathcal{C}}$.

In-Connected and Out-Connected Processes

We define now the set of *in-connected* processes as follows: (1) Every process in $\overset{*}{\mathcal{C}}$ is in-connected, and (2) a process p is in-connected if there exists a process $q \in \overset{*}{\mathcal{C}}$ for which $q \rightsquigarrow p$. Formally:

$$\textit{in-connected} = \{p \in \Pi : p \in \overset{*}{\mathcal{C}} \vee \exists q \in \overset{*}{\mathcal{C}} : q \rightsquigarrow p\}$$

Similarly, the set of *out-connected* processes is defined as follows: (1) Every process in $\overset{*}{\mathcal{C}}$ is out-connected, and (2) a process p is out-connected if there exits a process $q \in \overset{*}{\mathcal{C}}$ for which $p \rightsquigarrow q$. Formally:

$$\textit{out-connected} = \{p \in \Pi : p \in \overset{*}{\mathcal{C}} \vee \exists q \in \overset{*}{\mathcal{C}} : p \rightsquigarrow q\}$$

Note also that the assumption $|\overset{*}{\mathcal{C}}| \geq \lceil \frac{(n+1)}{2} \rceil$ makes $\rightsquigarrow$ relations harmless to the system.

Connectedness-based $\Diamond\mathcal{P}(om)$ Failure Detector

The *range* of the $\Diamond\mathcal{P}(om)$ failure detector is $\mathcal{R} = 2^{\Pi} \times \{\text{TRUE}, \text{FALSE}\}$. Informally it consists of a set of out-connected processes as well as a Boolean flag indicating whether the process considers itself as in-connected. Observe that such a failure detector is a trust based one instead of a suspicion based one.

A $\Diamond\mathcal{P}(om)$ *failure detector history* H is a function $H : \Pi \times T \mapsto \mathcal{R}$. Intuitively, $H(p,t) = (\{p,q,r\}, \text{FALSE})$ for example means that at time t process p considers processes p, q and r as out-connected and does not consider itself as in-connected. We denote by H_1 and H_2 the projection of the tuple returned by H to its first and second element, i.e., $H_1(p,t) = \{p,q,r\}$ and $H_2(p,t) = \text{FALSE}$ respectively in the previous example.

We formally define the class of *eventually perfect failure detectors for the omission model*, $\diamond\mathcal{P}(om)$, as the set of all failure detectors satisfying the following three properties:

- *In-connectedness*. Eventually every process will permanently consider itself as in-connected iff it is in-connected. Formally:

$$\forall F : \forall p \in \Pi : p \in \textit{in-connected} \Leftrightarrow \exists t_1 \in T : \forall t_2 \in T : t_2 > t_1 : H_2(p, t_2) = \text{TRUE}$$

- *Strong Completeness*. Every process that is not out-connected will not be permanently considered as out-connected by any in-connected process. Formally:

$$\forall F : \forall p \in \textit{in-connected} : \forall q \notin \textit{out-connected} : \forall t_1 \in T : \exists t_2 \in T : t_2 > t_1 : q \notin H_1(p, t_2)$$

- *Eventual Strong Accuracy*. Eventually every process that is out-connected will be permanently considered as out-connected by every in-connected process. Formally:

$$\forall F : \forall p \in \textit{in-connected} : \forall q \in \textit{out-connected} : \exists t_1 \in T : \forall t_2 \in T : t_2 > t_1 : q \in H_1(p, t_2)$$

This specification looks for in-connected processes to agree on the set of out-connected processes. For that, a process must determine if it is in-connected (In-connectedness). Due to timing and buffering attacks, some processes may behave as if they were out-connected without really being it, so it is not possible for in-connected processes to agree on the exact set of out-connected processes. Alternatively, Eventual Strong Accuracy ensures that out-connected processes will eventually be trusted forever. Finally, Strong Completeness avoids trusting forever a process that is not out-connected (be it *pseudo*-out-connected or omissive).

Our definition of an eventually perfect failure detector results naturally, i.e., at no additional cost, from the uniform (periodical, all-to-all) communication pattern used in our failure detection algorithm. We use this communication pattern in order to avoid side channel attacks (see Section VI) that otherwise an adversary could try in Byzantine environments. Said this, we guess that in a non-Byzantine omissive setting a failure detector of type $\diamond\mathcal{S}$ or Ω could be strong enough to solve consensus, as it is the case in the crash failure model.

IV. $\diamond\mathcal{P}(om)$-BASED CONSENSUS IN TRUSTEDPALS

We now focus our attention on the consensus layer of the TrustedPals architecture (see Fig. 4), in which we implement uniform consensus (as defined in Section II-E) using the failure detector class $\diamond\mathcal{P}(om)$ defined in the previous section. The consensus algorithm itself is *asynchronous*, meaning that it tolerates arbitrary phases of asynchrony. We first give the consensus algorithm and then prove its correctness.

{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) **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}

(12) **if** $p = c_p$ **then**

(13) **wait until** (**not** ($I_am_InConnected_p$) **or** (for $\lceil \frac{(n+1)}{2} \rceil$ processes q: received $(q, r_p, estimate_q, ts_q)$ from q))

(14) **if** *for* $\lceil \frac{(n+1)}{2} \rceil$ *processes q: received* $(q, r_p, estimate_q, ts_q)$ *from* q **then**

(15) $success_p \leftarrow$ TRUE

(16) $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$

(17) $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$

(18) $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$

(19) send $(p, r_p, estimate_p)$ to all

(20) **else**

(21) $success_p \leftarrow$ FALSE

(22) send (p, r_p, NEXT) to all

(23) **Phase 3:** {All processes wait for the new estimate proposed by the coordinator}

(24) **wait until** (**not** ($I_am_InConnected_p$) **or** $(c_p \in \Pi - OutConnected_p)$ **or** received [$(c_p, r_p, estimate_{c_p})$ **or** (c_p, r_p, NEXT)] from c_p)

(25) **if** *received* $(c_p, r_p, estimate_{c_p})$ *from* c_p **then**

(26) $estimate_p \leftarrow estimate_{c_p}$

(27) $ts_p \leftarrow r_p$

(28) send (p, r_p, ACK) to c_p

(29) **else**

(30) send (p, r_p, NACK) to c_p

(31) **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}

(32) **if** $(p = c_p)$ **and** $(success_p = \text{TRUE})$ **then**

(33) **wait until** [**not** ($I_am_InConnected_p$) **or** for each process q: (*received* (q, r_p, ACK) **or** *received* (q, r_p, NACK) **or** $q \in \Pi - OutConnected_p$)]

(34) **if** *for* $\lceil \frac{(n+1)}{2} \rceil$ *processes q:received* (q, r_p, ACK) **then** *R-broadcast*$(p, r_p, estimate_p, decide)$

Fig. 5. Solving consensus in the omission model using $\Diamond\mathcal{P}(om)$: main algorithm.

{If p R-delivers a decide message, p decides accordingly}
(35) **when** R-deliver(q, r_q, *estimate*$_q$, DECIDE) **do**
(36) **if** *state*$_p$ = *undecided* **then**
(37) *decide*(*estimate*$_q$)
(38) *state*$_p$ ← *decided*

Fig. 6. Solving consensus in the omission model using $\Diamond\mathcal{P}(om)$: adopting the decision.

A. Consensus Algorithm

Figs. 5 and 6 present an algorithm solving consensus in the omission model using $\Diamond\mathcal{P}(om)$. It is an adaptation of the well-known $\Diamond\mathcal{S}$-based Chandra-Toueg consensus algorithm [7] (which also works with a $\Diamond\mathcal{P}$ failure detector). The use of $\Diamond\mathcal{P}(om)$ by every process p is modeled by means of the following two variables: a Boolean variable *I_am_InConnected*$_p$ which provides the In-connectedness property, and a set *OutConnected*$_p$ which provides the Strong Completeness and Eventual Strong Accuracy properties.

The algorithm is based on the rotating coordinator paradigm. It executes in rounds, and each round is coordinated by a unique process, which tries to impose a value to the rest of participants. If it succeeds, then it takes a decision and reliably broadcasts [50] it to all processes, which adopt it. Otherwise, i.e., if the current coordinator is suspected, then processes advance to the next round. Each round is divided in four phases: a voting phase, a proposition phase, an acknowledgement phase, and a (potential) decision phase. An adequate use of the $\Diamond\mathcal{P}(om)$ failure detector ensures that, if not earlier, eventually a process that is well-connected will succeed in imposing a value and thus will decide.

We now comment on the modifications required to adapt the original Chandra-Toueg 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.
- 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.
- 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.

When a process p sends a message m to another process q, the following relaying 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 actual destination is another process q, p forwards m to all processes (except the process from which p has received m and p itself). This approach can take advantage of the underlying all-to-all implementation of the $\Diamond\mathcal{P}(om)$ failure detector without generating any extra message apart from the periodical all-to-all communication pattern of the failure detector, as we will see in the next section.

B. Correctness Proof

Since the algorithm is very similar to the one proposed by Chandra and Toueg [7], we only sketch the correctness proof here. 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 set $I_am_InConnected_p$ to FALSE. On the other hand, if p is in-connected, it will eventually receive a majority of estimates since by definition there is a majority of well-connected 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 $\Diamond\mathcal{P}(om)$ 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 $\Diamond\mathcal{P}(om)$ no coordinator blocks forever in the **wait** instruction of Phase 4.

By the previous facts, eventually some well-connected process c will coordinate a round in which:

- In Phase 2, the coordinator c will receive a majority of estimates, because $I_am_InConnected_c$ will be permanently set to TRUE (by the properties of $\Diamond\mathcal{P}(om)$) and there is a majority of well-connected processes in the system. Hence, c will send a valid estimate to all processes at the end of Phase 2.
- In Phase 3, every well-connected process p will receive c's valid estimate, because $I_am_InConnected_p$ will be permanently set to TRUE and c will be permanently in

OutConnected$_p$ (by the properties of $\Diamond\mathcal{P}(om)$). Hence, p will send an ACK message to c at the end of Phase 3.

- In Phase 4, the coordinator c will receive a majority of ACK messages, because *I_am_InConnected*$_c$ will be permanently set to TRUE and all well-connected processes will be permanently in *OutConnected*$_c$ (by the properties of $\Diamond\mathcal{P}(om)$) and there is a majority of well-connected processes in the system. Hence, c will R-broadcast the decision, and every in-connected process will eventually decide.

V. FAILURE DETECTION IN TRUSTEDPALS

Recall from Section II that at the transport layer TrustedPals merely assumes a partially synchronous system model. On top of this model, we now explain how to implement a failure detector of class $\Diamond\mathcal{P}(om)$, i.e., we focus on the failure detection layer of the TrustedPals architecture (see Fig. 4).

The failure detection algorithm presented in this section not only satisfies the properties defined in Section III, but also builds a basis for the consensus algorithm of Section IV. Specifically, our failure detector determines the connectivity relations defined in Section III. The algorithm is based on heartbeat messages that every process sends periodically to the rest of processes. This schema provides a kind of delayed message forwarding, which enables indirect communication of information attached to heartbeat messages. At the same time, it provides the support for piggybacking consensus level messages, as we will see in Section VI.

A. Failure Detector Algorithm

Figs. 7, 8 and 9 present an algorithm implementing $\Diamond\mathcal{P}(om)$ according to the properties defined in Section III. The algorithm provides to every process p a set of out-connected processes, *OutConnected*$_p$ and a Boolean variable, *I_am_InConnected*$_p$. The set *OutConnected*$_p$ will eventually and permanently contain all the out-connected processes in case p is in-connected. Regarding the *I_am_InConnected*$_p$ variable, it will be TRUE if p is in-connected.

The algorithm is based on the periodical communication of heartbeat messages from every process to the rest of processes. Roughly speaking, heartbeat messages carry information about connectivity among processes. When a process p receives a heartbeat message, it uses the connectivity information to update its perception of the connectivity of the rest of processes. This information, together with p's own connectivity, gives p a view of the current system connectivity, which will be propagated to the rest of processes attached to

```
(1)  Procedure main()
(2)    OutConnected_p ← Π
(3)    I_am_InConnected_p ← TRUE
(4)    forall q ∈ Π − {p} do
(5)      Δ_p(q) ← default time-out interval        {Δ_p(q) denotes the duration of p's time-out interval for q}
(6)      next_send_p[q] ← 1                        {sequence number of the next message sent to q}
(7)      next_receive_p[q] ← 1                     {sequence number of the next message expected from q}
(8)      Buffer_p[q] ← ∅
(9)    forall q ∈ Π do
(10)     forall u ∈ Π do M_p[q][u] ← 1             {M_p[q][u] = 0 means that q has not received at least one message from u}
(11)     Version_p[q] ← 0                          {Version_p contains the version number for every row of M_p}
(12)   UpdateVersion_p ← FALSE

(13) || Task 1: repeat periodically                {Sending heartbeats}
(14)   if UpdateVersion_p then                     {p's row has changed}
(15)     Version_p[p] ← Version_p[p] + 1
(16)     UpdateVersion_p ← FALSE
(17)   forall q ∈ Π − {p} do
(18)     send (ALIVE, p, next_send_p[q], M_p, Version_p) to q      {sends a heartbeat}
(19)     next_send_p[q] ← next_send_p[q] + 1      {p updates its sequence number for q}

(20) || Task 2: repeat periodically                {Checking time-outs}
(21)   if ( p did not receive (ALIVE, q, next_receive_p[q], M_q, Version_q)
            from q ≠ p during the last Δ_p(q) ticks of p's clock ) then   {next message not received timely}
(22)     if M_p[p][q] = 1 then
(23)       Δ_p(q) ← Δ_p(q) + 1
(24)       M_p[p][q] ← 0                           {the potential omission is reflected in M_p}
(25)       UpdateVersion_p ← TRUE
(26)       call update_Connectivity()

(27) || Task 3: when receive (ALIVE, q, c, M_q, Version_q) for some q   {Processing msgs. in order}
(28)   insert (ALIVE, q, c, M_q, Version_q) into Buffer_p[q]
(29)   while (ALIVE, q, next_receive_p[q], M_q, Version_q) ∈ Buffer_p[q] do   {it is the next expected message from q}
(30)     call deliver_next_message(q, M_q, Version_q)                        {the message is delivered}
(31)     remove (ALIVE, q, M_q, next_receive_p[q], Version_q) from Buffer_p[q]
(32)     next_receive_p[q] ← next_receive_p[q] + 1
(33)   if Buffer_p[q] = ∅ then
(34)     M_p[p][q] ← 1                             {so far, p has received all messages from q}
(35)     UpdateVersion_p ← TRUE
(36)   if M_p has changed then call update_Connectivity()
```

Fig. 7. $\Diamond\mathcal{P}(om)$ in the omission model: main algorithm.

p's subsequent heartbeat messages. Next we explain in detail how our algorithm implements this approach.

Every process p has a matrix M_p of $n \times n$ elements representing connectivity information ($\rightarrow$ relations between every pair of processes). Every heartbeat message from p carries the

```
(37) Procedure update_Connectivity()
(38)   A_p ← (M_p)^n                                   {A_p is the n-th power of the M_p matrix}
(39)   forall u, v ∈ Π do
(40)     if A_p[u][v] > 0 then  A_p[u][v] ← 1
(41)   Out ← ∅
(42)   forall q ∈ Π do
(43)     if (Σ_{i=0}^{n-1} A_p[i][q] ≥ ⌈(n+1)/2⌉) then  Out ← Out ∪ {q}
(44)   OutConnected_p ← Out
(45)   I_am_InConnected_p = (Σ_{i=0}^{n-1} A_p[p][i] ≥ ⌈(n+1)/2⌉)
```

Fig. 8. $\Diamond\mathcal{P}(om)$ in the omission model: procedure update_Connectivity().

```
(46) Procedure deliver_next_message()
(47)   forall v ∈ Π do  M_p[q][v] ← M_q[q][v]          {q's row of M_q is systematically copied into M_p}
(48)   Version_p[q] ← Version_q[q]
(49)   forall u ∈ Π − {p, q} do
(50)     if Version_q[u] > Version_p[u] then             {q's information about u is more recent than p's}
(51)       forall v ∈ Π do  M_p[u][v] ← M_q[u][v]
(52)       Version_p[u] ← Version_q[u]
```

Fig. 9. $\Diamond\mathcal{P}(om)$ in the omission model: procedure deliver_next_message().

matrix M_p and a sequence number used to detect message omissions. Received messages are buffered to be delivered in FIFO order. When a heartbeat from process q is received by process p, p updates M_p according to the state of its input channels and with the received M_q. Actually, M_p represents the transposed adjacency matrix, a (0,1)-matrix of a directed graph, where the value of the element $M_p[p][q]$ shows if there is an arc from q to p. M_p has the information needed to calculate the set of out-connected processes and the value of the *I_am_InConnected*$_p$ variable. The algorithm calculates powers of the adjacency matrix to find paths of any length between processes, which correspond to transitive relations $q \rightsquigarrow p$, and therefore representing indirect communication paths from q to p. The set of out-connected processes, along with the *I_am_InConnected*$_p$ variable, is computed in the *update_Connectivity*() procedure (Fig. 8), which is called every time a value of the matrix M_p is changed. Observe that it is important for a process p to check its own in-connectivity to verify the validity of the information contained in M_p. The in-connectivity condition of p (more than $\lfloor n/2 \rfloor$ processes communicate properly with it) is checked in the *update_Connectivity*() procedure too, and its value is output to the *I_am_InConnected*$_p$ variable.

In the algorithm, every process p executes three tasks:

- In Task 1 (line 13), p periodically sends to the rest of processes a heartbeat message

including the matrix M_p and a unique sequence number.

- In Task 2 (line 20), if p does not receive the expected message from a process q (according to the *next_receive*$_p[q]$ sequence number) in the expected time, the value of $M_p[p][q]$ is set to 0.
- In Task 3 (line 27), received messages are processed. The messages p receives from another process q are inserted in a FIFO buffer *Buffer*$_p[q]$ (line 28), and delivered following the sequence number *next_receive*$_p[q]$. Once delivered the next expected message from a process q, the condition of empty buffer means that there is no message left from q, so $M_p[p][q]$ is set to 1.

The procedure *deliver_next_message*$()$ (Fig. 9) 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 associated version number is higher.

B. Correctness Proof

We now show that the algorithm of Figs. 7, 8 and 9 implements $\Diamond\mathcal{P}(om)$ in the omission model.

Observation 1: At every process p, the matrix M_p is updated with its own connectivity information and with the matrices M_q received in the heartbeat messages. The updated M_p and its version number *Version*$_p$ are sent with p's next heartbeat message. The local delay in process p to send M_p and *Version*$_p$ is bounded in the algorithm by the period of Task 1 of p, which is finite if p is eventually synchronous and has not crashed. This way, a schema equivalent to the relaying mechanism described in Section III is obtained.

Lemma 1: $\forall p, q \in \Pi$, iff $q \rightsquigarrow p$, then eventually and permanently $(M_p)^n[p][q] \geq 1$.

If $q \rightsquigarrow p$, then, by definition of $\rightsquigarrow$, eventually and permanently there is a reliable and *timely*[1] path with no omission of any length from q to p. By Task 1 q sends periodically a heartbeat message including its updated M_q and *Version*$_q[q]$ to the rest of processes. Every process r receiving by Task 3 a new *Version*$_q[q]$ will update from M_q, in the procedure *deliver_next_message*$()$, the row q of M_r as well as *Version*$_r[q]$. Process r will update matrix

[1]Here by *timely* we mean that local clock timeouts do not expire.

M_r too with its own connectivity information: by Task 3 of r, $M_r[r][q]$ is set to 1 every time *Buffer*$_r[q]$ becomes empty; by Task 2 $M_r[r][q]$ is set to 0 when the next expected message from q is not received timely by r. By Observation 1, $M_r[r][q]$ and *Version*$_r[r]$ are propagated to every process s if s is eventually synchronous and has not crashed. (Note that, as a particular case, r or s may be p.) When some message is delivered by Task 3 of p, by the procedure *deliver_next_message*$()$, p will update M_p and calculate $(M_p)^n$ if some value in M_p has changed. By the definition of the relation $\rightsquigarrow$, If $q \rightsquigarrow p$ then $(M_p)^n[p][q]$ will be evaluated eventually and permanently to a positive value, otherwise, if not $q \rightsquigarrow p$, $(M_p)^n[p][q]$ will not be set to 1 permanently because, by definition, in every possible path from q to p there will be two processes, r and s, such that not $r \rightarrow s$ (r and s could be q and p), and therefore, $(M_p)^n[p][q] = 0$.

Lemma 2: $\forall p \in$ *in-connected*, $\forall q \in$ *out-connected*, eventually and permanently $q \in$ *OutConnected*$_p$.

Proof: By definition of out-connected process, there is some well-connected process r such that $q \rightsquigarrow r$. By definition of in-connected process, there is some well-connected process s such that $s \rightsquigarrow p$. By transitivity, for every well-connected process u, $q \rightsquigarrow u$, and in particular $q \rightsquigarrow s$. By Lemma 1, $(M_s)^n[s][q] \geq 1$. By the procedure *deliver_next_message*$()$, s will update M_s copying all the rows from at least the rest of well-connected processes. As a consequence of that, since $|\overset{*}{\mathcal{C}}| > \lfloor n/2 \rfloor$, column q of $(M_s)^n$ will include eventually and permanently more than $\lfloor n/2 \rfloor$ positive values. Again by Lemma 1 and the procedure *deliver_next_message*$()$, column q of $(M_p)^n$ will have eventually and permanently a positive value for more than $\lfloor n/2 \rfloor$ processes. As a consequence, according to the procedure *update_Connectivity*$()$, q will be permanently included in the set *OutConnected*$_p$. ■

Lemma 3: $\forall p \in$ *in-connected*, $\forall q \notin$ *out-connected*, q is not permanently in *OutConnected*$_p$.

Proof: Since q is not out-connected, it does not exist a well-connected process r such that $q \rightsquigarrow r$. By Lemma 1, $(M_p)^n[r][q] \geq 1$ only when q receives messages timely from r, however, since q is not out-connected, this will not occur permanently. As a consequence, since $|\overset{*}{\mathcal{C}}| > \lfloor n/2 \rfloor$, the number of processes s such that $(M_p)^n[s][q] \geq 1$ will not be permanently greater than $\lfloor n/2 \rfloor$, and by the procedure *update_Connectivity*$()$, q will not be permanently included in the set *OutConnected*$_p$. ■

Lemma 4: $\forall p \in \Pi$, iff $p \in$ *in-connected*, then eventually and permanently *I_am_InConnected*$_p =$ TRUE.

Proof: If $p \in$ *in-connected*, $\forall r \in \overset{*}{\mathcal{C}}$, $r \rightsquigarrow p$. By Lemma 1, and following a similar

reasoning to the proof of Lemma 2, now applied to row p of $(M_p)^n$, iff $p \in$ *in-connected* the procedure *update_Connectivity*$()$ will eventually and permanently set *I_am_InConnected*$_p$ to TRUE (line 45). ■

Theorem 1: The algorithm of Figs. 7, 8 and 9 implements $\Diamond\mathcal{P}(om)$ in the omission model.

Proof: The strong completeness, eventual strong accuracy, and in-connectivity properties of $\Diamond\mathcal{P}(om)$ are satisfied by Lemmas 3, 2, and 4 respectively. ■

VI. INTEGRATING FAILURE DETECTION AND CONSENSUS SECURELY

Most properties of SMC have their direct counterparts in properties provided by the consensus abstraction used in TrustedPals. One notable difference is the property of SMC-Privacy, which relies on subtle issues not relevant to fault-tolerant synchronization. As an example, it is possible to successfully attack TrustedPals if the attacker can distinguish different message types on the network. Recall that every smartcard has to process messages from the consensus protocol and messages from the failure detector (see Fig. 4). Therefore it is important to integrate the consensus and failure detector algorithms securely in TrustedPals.

A. Types of Messages

In TrustedPals there are two types of messages sent over the channel: messages by the consensus algorithm and failure detector messages. We call the former *protocol messages* and the latter *heartbeats*. Heartbeats are time critical, i.e., they should not be delayed by the transport layer, while protocol messages are asynchronous, i.e., eventual delivery is sufficient for them.

The idea of TrustedPals is to use heartbeats as the *transport mechanism* for protocol messages, providing an implicit relaying mechanism to the consensus layer. Every heartbeat has a small fixed-size message field called the *payload*. Similar to network transport protocols (like IP) protocol messages are inserted into the payload of heartbeats when they are sent. If a protocol message is larger than the size of the payload, it is fragmented into smaller parts which are sent one after the other. Similar to the fragmentation mechanism in IP, unique identifiers and sequence numbers allow to piece together the fragments at the destination in the correct order. If there is no protocol message to be sent, the payload of a heartbeat can remain *empty*. In this way, we achieve that TrustedPals generates *fixed size* messages in *fixed* (periodic) time intervals.

Observe that the use of heartbeats as the transport mechanism for protocol messages has a negative impact on the performance of the consensus algorithm, since the consensus messages are delayed until messages of the failure detection layer are sent. This is the price to pay for hiding the communication pattern of the consensus algorithm to the adversary.

B. Avoiding Message and Traffic Analysis

Traffic analysis refers to an attack technique which tries to derive information about messages by simply analysing the variation of the times in which they are sent and received. Since we send protocol messages “within” heartbeats we increase the difficulty for the attacker to distinguish an “empty” heartbeat from a “full” heartbeat by only looking at the timing of traffic. Since heartbeats are sent in an all-to-all pattern, it is also hard to distinguish which process is sending protocol messages to which other process. This approach ensures *unobservability* regarding protocol messages, a notion known from the area of privacy-enhancing techniques [47], [48].

Of course, the attacker could simply look into the contents of a heartbeat to discern a full from an empty message. That is why we employ cryptography on the channel. We implement a secure channel satisfying confidentiality, integrity and authenticity using standard techniques from cryptography [51]. The idea is to use the assumed keying material within the security modules to encrypt all messages and to use message authentication codes or digital signatures to ensure the integrity and authenticity of the channel.

Important from a message-analysis point of view is that all heartbeats are indistinguishable from each other. As mentioned above, all heartbeats messages have the same length and if they are encrypted they will ideally look like random data. Hence, from just analysing the contents of a heartbeat it is impossible to distinguish a full from an empty heartbeat. Note that information about source and destination of a heartbeat must be sent unencrypted to allow routing. However, this information must also be stored within the encrypted part of the message to ensure authenticity.

VII. Conclusions

We have presented a modular redesign of TrustedPals, a smartcard-based security framework capable of efficiently solving Secure Multiparty Computation. The framework is based on a two-part architecture. One part represents the untrusted system, consisting of untrusted,

Byzantine hosts. In the other part, representing the trusted system, security modules reduce the security problem to a fault-tolerant consensus among smartcards.

The modular redesign allows TrustedPals to face the consensus problem in the general-omission failure model, which is more benign than the Byzantine model. Besides that, the trusted system has to deal with attacks which cannot be filtered by smartcards, specifically timing attacks and buffering attacks, resulting in a system model that includes asynchronous processes and/or channels. According to that, we model classes of processes and types of communication among them, and give a novel definition of the eventually perfect failure detector class for the omission failure model. The new failure detector properties are based on process connectedness rather than on process correctness. We have proposed a failure detector algorithm which assumes a majority of well-connected processes, and a consensus algorithm using it. Interestingly, the consensus algorithm is an adaptation of the classical $\Diamond\mathcal{S}$-based Chandra-Toueg algorithm for the crash model.

Another relevant aspect of the redesign is the integration of failure detection and consensus into the TrustedPals framework. Since the failure detector follows a heartbeat-based, all-to-all communication pattern, TrustedPals uses heartbeats as the transport mechanism for consensus messages. This approach ensures unobservability. Conceptually, the system is reasonably secure against almost all practical attacks.

Our algorithms can be improved with respect to efficiency. In particular, our implementation of $\Diamond\mathcal{P}(om)$ can be modified such that it results in Ω by omitting the all-to-all message exchange pattern, saving a substantial amount of messages. However, the decision to choose $\Diamond\mathcal{P}(om)$ was deliberate since the integration into TrustedPals makes all-to-all communication necessary anyway to protect against side channel analysis that could endanger security. Therefore, any such efficiency improvement would be futile in practical systems. Nevertheless, we consider that determining the weakest failure detector for solving consensus in this system is an interesting open question which deserves further research. Also, the space requirements of the failure detector messages (mainly the matrix of bits) can be compressed substantially by special encoding techniques in practice.

As future work, we intend to implement the approach and perform practical experiments with the system. Within the failure detector implementation, the size of the payload field will be an interesting parameter to choose. 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.

On the theoretical side it would be interesting to study the minimal storage and communication effort necessary to solve consensus in our model, since we use unbounded buffers in our implementation and the bit complexity of the messages we use is also rather high.

References

[1] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*. IEEE, 1982, pp. 160–164.

[2] M. Fort, F. C. Freiling, L. D. Penso, Z. Benenson, and D. Kesdogan, "Trustedpals: Secure multiparty computation implemented with smart cards," in *ESORICS*, ser. LNCS, vol. 4189. Springer, 2006, pp. 34–48.

[3] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] N. Leavitt, "Will proposed standard make mobile phones more secure?" *IEEE Computer*, vol. 38, no. 12, pp. 20–22, 2005.

[5] certgate GmbH, "certgate Smart Card," http://www.certgate.com/web_en/products/smartcardmmc.html, 2008.

[6] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[7] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[8] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," *ACM Comput. Surv.*, vol. 43, no. 2, p. 9, 2011.

[9] D. Malkhi and M. K. Reiter, "Unreliable intrusion detection in distributed computations," in *CSFW*. IEEE Computer Society, 1997, pp. 116–125.

[10] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Byzantine fault detectors for solving consensus," *Comput. J.*, vol. 46, no. 1, pp. 16–35, 2003.

[11] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *EDCC*, ser. LNCS, vol. 1667. Springer, 1999, pp. 71–87.

[12] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating failure detection: From crash to Byzantine failures," in *Ada-Europe*, ser. LNCS, vol. 2361. Springer, 2002, pp. 24–50.

[13] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for Byzantine fault detection," in *2nd Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[14] A. Haeberlen and P. Kuznetsov, "The fault detection problem," in *OPODIS*, ser. LNCS, vol. 5923. Springer, 2009, pp. 99–114.

[15] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[16] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *NSDI*. USENIX Association, 2009, pp. 153–168.

[17] H. Moniz, N. F. Neves, and M. Correia, "Turquois: Byzantine consensus in wireless ad hoc networks," in *DSN*. IEEE, 2010, pp. 537–546.

[18] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, 1985.

[19] M. Herlihy and J. D. Tygar, "How to make replicated data secure," in *CRYPTO*, ser. LNCS, vol. 293. Springer, 1987, pp. 379–391.

[20] C. Delporte-Gallet, H. Fauconnier, and F. C. Freiling, "Revisiting failure detection and consensus in omission failure environments," in *ICTAC*, ser. LNCS, vol. 3722. Springer, 2005, pp. 394–408.

[21] C. Delporte-Gallet, H. Fauconnier, A. Tielmann, F. C. Freiling, and M. Kilic, “Message-efficient omission-tolerant consensus with limited synchrony,” in *IPDPS*. IEEE, 2009, pp. 1–8.

[22] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi, “Failure detectors in omission failure environments,” Cornell University, Computer Science Department, Tech. Rep. TR96-1608, 1996.

[23] ——, “Failure detectors in omission failure environments,” in *PODC*, 1997, p. 286.

[24] Ö. Babaoglu, R. Davoli, and A. Montresor, “Group communication in partitionable systems: Specification and algorithms,” *IEEE Trans. Software Eng.*, vol. 27, no. 4, pp. 308–336, 2001.

[25] N. Santoro and P. Widmayer, “Agreement in synchronous networks with ubiquitous faults,” *Theor. Comput. Sci.*, vol. 384, no. 2-3, pp. 232–249, 2007.

[26] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo, “Randomization can be a healer: Consensus with dynamic omission failures,” in *DISC*, ser. LNCS, vol. 5805. Springer, 2009, pp. 63–77.

[27] P. D. MacKenzie, A. Oprea, and M. K. Reiter, “Automatic generation of two-party computations,” in *ACM Conference on Computer and Communications Security*. ACM, 2003, pp. 210–219.

[28] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, “Fairplay - secure two-party computation system,” in *USENIX Security Symposium*. USENIX, 2004, pp. 287–302.

[29] Y. Lindell and B. Pinkas, “An efficient protocol for secure two-party computation in the presence of malicious adversaries,” in *EUROCRYPT*, ser. LNCS, vol. 4515. Springer, 2007, pp. 52–78.

[30] V. Kolesnikov, “Gate evaluation secret sharing and secure one-round two-party computation,” in *ASIACRYPT*, ser. LNCS, vol. 3788. Springer, 2005, pp. 136–155.

[31] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh, “Secure function evaluation with ordered binary decision diagrams,” in *ACM Conference on Computer and Communications Security*. ACM, 2006, pp. 410–420.

[32] M. Burkhart, M. Strasser, D. Many, and X. A. Dimitropoulos, “Sepia: Privacy-preserving aggregation of multi-domain network events and statistics,” in *USENIX Security Symposium*. USENIX Association, 2010, pp. 223–240.

[33] G. Avoine and S. Vaudenay, “Optimal fair exchange with guardian angels,” in *WISA*, ser. LNCS, vol. 2908. Springer, 2003, pp. 188–202.

[34] G. Avoine, F. C. Gärtner, R. Guerraoui, and M. Vukolic, “Gracefully degrading fair exchange with security modules,” in *EDCC*, ser. LNCS, vol. 3463. Springer, 2005, pp. 55–71.

[35] M. Ben-Or, R. Canetti, and O. Goldreich, “Asynchronous secure computation,” in *STOC*, 1993, pp. 52–61.

[36] O. Goldreich, S. Micali, and A. Wigderson, “How to play ANY mental game or a completeness theorem for protocols with honest majority,” in *STOC*. ACM, 1987, pp. 218–229.

[37] D. Chaum, I. Damgård, and J. van de Graaf, “Multiparty computations ensuring privacy of each party’s input and correctness of the result,” in *CRYPTO*, ser. LNCS, vol. 293. Springer, 1987, pp. 87–119.

[38] M. Ben-Or, B. Kelmer, and T. Rabin, “Asynchronous secure computations with optimal resilience (extended abstract),” in *PODC*, 1994, pp. 183–192.

[39] M. Correia, P. Veríssimo, and N. F. Neves, “The design of a COTS-real-time distributed security kernel,” in *EDCC*, ser. LNCS, vol. 2485. Springer, 2002, pp. 234–252.

[40] P. Sousa, N. F. Neves, and P. Veríssimo, “Proactive resilience through architectural hybridization,” in *SAC*. ACM, 2006, pp. 686–690.

[41] O. Goldreich, “Secure multi-party computation,” Internet: http://www.wisdom.weizmann.ac.il/~oded/pp.html, 2002.

[42] L. Lamport, R. E. Shostak, and M. C. Pease, “The Byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[43] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, “Consensus in the presence of partial synchrony,” *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[44] V. Hadzilacos, “Issues of fault tolerance in concurrent computations,” Ph.D. dissertation, Harvard University, 1984, also published as Technical Report TR11-84.

[45] K. J. Perry and S. Toueg, “Distributed agreement in the presence of processor and communication faults,” *IEEE Trans. Software Eng.*, vol. 12, no. 3, pp. 477–482, 1986.

[46] P. R. Parvédy and M. Raynal, “Optimal early stopping uniform consensus in synchronous systems with process omission failures,” in *SPAA*. ACM, 2004, pp. 302–310.

[47] G. Danezis and C. Diaz, “A survey of anonymous communication channels,” Microsoft Research, Tech. Rep. MSR-TR-2008-35, 2008.

[48] A. Pfitzmann and M. Hansen, “Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management – A Consolidated Proposal for Terminology,” TU Dresden, Tech. Rep., 2008.

[49] F. Cristian and C. Fetzer, “The timed asynchronous distributed system model,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 642–657, 1999.

[50] V. Hadzilacos and S. Toueg, *Distributed systems*, 2nd ed. ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant broadcasts and related problems, pp. 97–145.

[51] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.