TrustedPals: Secure Multiparty Computation Implemented with Smart Cards

Zinaida Benenson[1], Milan Fort[2], Felix Freiling[3],
Dogan Kesdogan[2], and Lucia Draque Penso[3]

[1] Department of Information Technology, Uppsala University,
SE-751 05 Uppsala, Sweden
[2] Computer Science Department, RWTH Aachen University,
D-52056 Aachen, Germany
[3] Computer Science Department, University of Mannheim,
D-68131 Mannheim, Germany

Abstract. We study the problem of *Secure Multi-party Computation* (*SMC*) in a model where individual processes contain a tamper-proof security module, and introduce the *TrustedPals* framework, an efficient smart card based implementation of *SMC* for any number of participating entities in such a model. Security modules can be trusted by other processes and can establish secure channels between each other. However, their availability is restricted by their host, that is, a corrupted party can stop the computation of its own security module as well as drop any message sent by or to its security module. We show that in this model SMC can be implemented by reducing it to a fault-tolerance problem at the level of security modules. Since the critical part of the computation can be executed locally on the smart card, we can compute any function securely with a protocol complexity which is polynomial only in the number of processes (that is, the complexity does not depend on the function which is computed), in contrast to previous approaches.

1 Introduction

Motivation. The problem of *Secure Multi-party Computation* (*SMC*, sometimes also referred to as *Secure Function Evaluation*), is one of the most fundamental problems in security. The setting is as follows: a set of n parties jointly wants to compute the result of an n-ary function F. Every party provides its own (private) input to this function but the inputs should remain secret to the other parties, except for what can be derived from the result of F. The problem is easy to solve if you assume the existence of a trusted third party (TTP) which collects the inputs, computes F and distributes the result to everyone. However, the problem is very challenging if you assume that there is no TTP available and parties can misbehave arbitrarily, that, they can send wrong messages or fail to send messages at all. Still, the protocol must correctly and securely compute F as if a TTP were available.

Having been initially proposed by Yao in 1982 [29], it got its first solution only in 1987, when Goldreich, Micali and Wigderson [16] showed that in a synchronous system with cryptography a majority of honest processes can simulate a centralized trusted third party. This was done by transforming the function F into a computation over a finite field and then showing that addition and multiplication in this finite field could be implemented securely using secret sharing and agreement protocols. It was also shown that a majority of honest processes was necessary for SMC.

All existing general solutions to SMC are based on the original idea of Goldreich, Micali and Wigderson [16]. Hence, the message complexity always depends on the function that is computed. For example, the most efficient solution to SMC we are aware of [18] requires communicating $O(m \cdot n^3)$ field elements (m is the number of multiplication gates in F) and at least $O(n^2)$ rounds of communication (in fact, the round complexity also depends on F). Thus, despite solutions, many practitioners have been prevented to attempting to implement general SMC due to lack of efficiency.

Recently, there has been an increasing interest in SMC which probably stems from the growing importance and the difficulty to implement fault-tolerance in combination with security in today's networks. In fact, in the concluding remarks on the COCA project, Zhou, Schneider and van Renesse [30] call to investigate practical secure multi-party computation.

Related Work. In 2003, MacKenzie, Oprea and Reiter [21] presented a tool which could securely compute a two-party function over a finite field of a specific form. Later, Malkhi *et al.* [22] presented *Fairplay*, a general solution of two-party secure computation. Both papers follow the initial approach proposed by Goldreich, Micali and Wigderson [16], that is, they make extensive use of compilers that translate the function F into one-pass boolean circuits. Iliev and Smith [19] report in yet unpublished work on performance improvements using trusted hardware. In this paper we revisit SMC in a similar model but using a different approach.

The approach we use in this paper was pioneered by Avoine and Vaudenay [4]. It assumes a synchronous model with no centralized TTP, but the task of jointly simulating a TTP is alleviated by assuming that parties have access to a local *security module* (Avoine and Vaudenay [4] call this a *guardian angel*). Recently, manufacturers have begun to equip hardware with such modules: these include for instance smart cards or special microprocessors. These are assumed to be tamper proof and run a certified piece of software. Examples include the Embedded Security Subsystem within the recent IBM Thinkpad or the IBM 4758 secure co-processor board [10]. A large body of computer and device manufacturers has founded the Trusted Computing Group (TCG) [28] to promote this idea. Security modules contain cryptographic keys so that they can set up secure channels with each other. However, they are dependant on their hosts to be able to communicate with each other.

Later, Avoine *et al.* [3] showed that, in a model with security modules, the fair exchange problem, an instance of SMC, can be reduced to an agreement

problem among security modules, which can itself be transformed to the consensus problem, a classical problem of fault-tolerant distributed computing. The reduction allows modular solutions to fair exchange, as the agreement abstraction can be implemented in different ways [9, 14]. The problem of SMC has not yet been investigated in this model.

Contributions. In this paper, we investigate the resilience and efficiency of SMC in the model of untrusted hosts and security modules. In this model the security modules and their communication network form a subnetwork with a more benign fault assumption, namely that of general omission [25]. In the general omission failure model processes may simply stop executing steps or fail to send or receive messages sent to them.

We extend the work by Avoine *et al.* [3] and derive a novel solution to SMC in a modular way: We show that SMC is solvable if and only if the problem of Uniform Interactive Consistency (UIC) is solvable in the network of security modules. UIC is closely related to the problem of Interactive Consistency [24], a classic fault-tolerance problem. From this equivalence we are able to derive a basic impossibility result for SMC in the new model: We show that UIC requires a majority of correct processes and from this can conclude that SMC is impossible in the presence of a dishonest majority. This shows that, rather surprisingly, adding security modules cannot improve the resilience of SMC. However, we prove that adding security modules can considerably improve the efficiency of SMC protocols. This is because the computation of F can be done locally within the security modules and does not affect the communication complexity of the SMC protocol. Therefore our solution to SMC which uses security modules requires only $O(n)$ rounds of communication and $O(n^3)$ messages. To the best of our knowledge, this is the first solution for which the message and round complexity do not depend on the function which is computed.

Furthermore, we give an overview of *TrustedPals*, a peer-to-peer implementation of the security modules framework using Java Card Technology enabled smart cards [8]. Roughly speaking, in the TrustedPals framework, F is coded as a Java function and is distributed within the network in an initial setup phase. After deployment, the framework manages secure distribution of the input values and evaluates F on the result of an agreement protocol between the set of security modules. To show the applicability of the framework, we implemented the approach of Avoine *et al.* [3] for fair exchange. To our knowledge, TrustedPals is the first practical implementation of SMC (1) for two *and more* processes and (2) which does not require a transformation into and a subsequent computation in a finite field. While still experimental, the TrustedPals framework is available for download [1].

Roadmap. We first present the model in Section 2. We then define the *security* problem of secure multi-party computation (SMC) and the *fault-tolerance problem* of uniform interactive consistency (UIC) in Sections 3 and 4. The security problems arising when using UIC within a SMC protocol are discussed in Section 5. We present the equivalence of SMC and UIC, in Section 6, and finally

describe the TrustedPals efficient framework in Section 7. Proofs are relegated to the appendix.

2 Model

2.1 Processes and channels

The system consists of a set of processes interconnected by a synchronous communication network with reliable secure bidirectional channels. Two processes connected by a channel are said to be adjacent.

A reliable secure channel connecting processes P and Q satisfies the following properties:

- (No Loss) No messages are lost during the transmission over the channel.
- (No Duplication) All messages are delivered at most once.
- (Authenticity) If a message is delivered at Q, then it was previously sent by P.
- (Integrity) Message contents are not tampered with during transmission, i.e., any change during transmission will be detected and the message will be discarded.
- (Confidentiality) Message contents remain secret from unauthorized entities.

In a synchronous network communication proceeds in rounds. In each round, a party first receives inputs from the user and all messages sent to it in the previous round (if any), processes them and may finally send some messages to other parties or give outputs to the user.

2.2 Untrusted hosts and security modules

The set of processes is divided into two disjoint classes: *untrusted hosts* (or simply *hosts*) and *security modules*. We assume that there exists a fully connected communication topology between the hosts, i.e., any two hosts are adjacent. We denote by n the number of hosts in the system. Furthermore, we assume that every host process H_A is adjacent to exactly one security module process M_A (there is a bijective mapping between security modules and hosts). In this case we say that H_A is *associated with* M_A (M_A is H_A's associated security module). We call the part of the system consisting only of security modules and the communication links between them the *trusted system* (see Fig. 1).

We call the part of the system consisting only of hosts and the communication links between them the *untrusted system*. The notion of association can be extended to systems, meaning that for a given untrusted system, the *associated trusted system* is the system consisting of all security modules associated to any host in that untrusted system.

In some definitions we use the term process to refer to both a security module and a host. We do this deliberately to make the definitions applicable both in the trusted and the untrusted system.

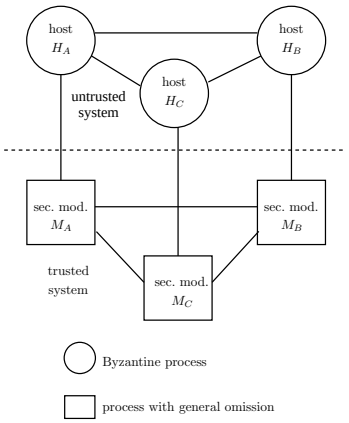

Fig. 1. Hosts and security modules.

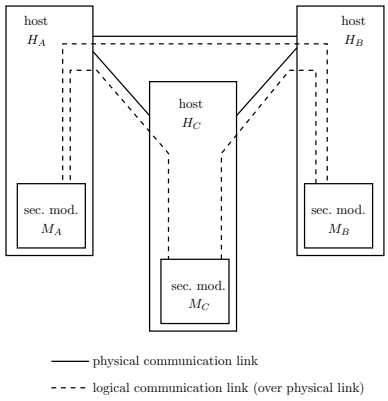

Fig. 2. Internet hosts with tamper-proof hardware corresponding to Fig. 1.

2.3 Relation to Systems with Trusted Hardware.

The model sketched above can be related to the setup in practice as follows: untrusted hosts model Internet hosts and their users, whereas security modules abstract tamper proof components of user systems (like smart cards, see Fig. 2). Intuitively, security modules can be trusted by other security modules or hosts, and hosts cannot be trusted by anybody. Hosts may be malicious, i.e., they may actively try to fool a protocol by not sending any message, sending wrong messages, or even sending the right messages at the wrong time.

Security modules are supposed to be cheap devices without their own source of power. They rely on power supply from their hosts. In principle, a host may cut off the power supply to its security module whenever he chooses, thereby preventing the security module from continuing to execute steps. Instead of doing this, a host may inhibit some or even *all* communication between its associated security module and the outside world.

2.4 Trust and adversary model

The setting described above is formalized using distinct failure models for different parts of the system. We assume that nodes in the untrusted system can act arbitrarily, i.e., they follow the Byzantine failure model [20]. In particular, the incorrect processes can act together according to some sophisticated strategy, and they can pool all information they possess about the protocol execution. We assume however that hosts are computationally bounded, i.e., brute force attacks on secure channels are not possible.

For the trusted system we assume the failure model of *general omission* [25], i.e., processes can crash or fail by not sending messages or not receiving messages.

A process is *faulty* if it does not correctly follow the prescribed protocol. In particular, a security module is faulty if it crashes or commits send or receive

omissions. Otherwise a process is said to be *correct*. In a system with n processes, we use t to denote a bound on the number of hosts which are allowed to be faulty.

3 Secure Multi-Party Computation

In *secure multi-party computation* (SMC), a set of processes $p_1, \ldots, p_n$, each starting with an input value x_i, wants to compute the result of a deterministic function F, i.e., $r = F(x_1, \ldots, x_n)$. Result r should be computed reliably and securely, i.e., as if they were using a trusted third party (TTP). This means that the individual inputs remain secret to other processes (apart from what is given away by r) and that malicious processes can neither prevent the computation from taking place nor influence r in favorable ways.

We assume that F is a well-known deterministic function with input domain X and output domain Y upon which all processes have agreed upon beforehand and that all correct processes jointly begin the protocol. We say that r is an *F-result* if r was computed using F. Since faulty processes cannot be forced to submit their input value, F may be computed using a special value $\perp \notin X$ instead of the input value of a faulty process.

Instead of defining SMC using a TTP [15], we now define SMC using a set of abstract properties.

Definition 1 (secure multi-party computation). *A protocol solves* secure multi-party computation (SMC) *if it satisfies the following properties:*

- *(SMC-Validity) If a process receives an F-result, then F was computed with at least the inputs of all correct processes.*
- *(SMC-Agreement) If some process p_i receives F-result r_i and some process p_j receives F-result r_j then $r_i = r_j$.*
- *(SMC-Termination) Every correct process eventually receives an F-result.*
- *(SMC-Privacy) Faulty processes learn nothing about the input values of correct processes (apart from what is given away by the result r and the input values of all faulty processes).*

From a security protocols perspective, the above definition can be considered slightly stronger than the usual (cryptographic) definitions of SMC since it demands that SMC-properties hold without any restriction. In the literature it is often stated that the probability of a violation of SMC-properties can be made arbitrarily small. We have chosen this stronger definition to simplify the presentation. We believe that definitions, theorems and proofs can be transferred into a probabilistic model with moderate effort.

The properties of SMC are best understood by comparing them to a solution based on a TTP. There, the TTP waits for the inputs of all n processes and computes the value of F on all those inputs which it received. Since all correct processes send their input value to the TTP, F is computed on at least those values, which motivates SMC-Validity. After computing F, the TTP sends the result back to all processes. Hence, all correct processes eventually receive that

result (SMC-Termination). Additionally, if a process receives a result from the TTP, then it will be the same result which any other process (whether correct or faulty) will receive. This motivates SMC-Agreement. SMC-Privacy is motivated by the fact that the TTP does all the processing and channels to the TTP are confidential: no information about other processes' input values leaks from this idealized entity, apart of what the result of F gives away when it is finally received by the processes.

4 Uniform Interactive Consistency

The problem of *Interactive Consistency* (IC) was introduced by Pease, Shostak and Lamport in 1980 [24]. It is one of the classical problems of reliable distributed computing since solutions to this problem can be used to implement almost any type of fault-tolerant service [27]. In this problem, every process starts with an initial value v_i. To solve the problem, the set of processes needs to agree on a vector D of values, one per process (Agreement property). Once vector D is output by process p, we say that p *decides* D. The i-th component of this vector should be v_i if p_i does not fail, and can be $\perp$ otherwise. IC is equivalent to the (also classic) *Byzantine Generals Problem* [20].

Definition 2 considers a version of IC with a stronger agreement property called *Uniform Agreement*. Uniform Agreement demands that *all* processes should decide the same (if they decide) — it does not matter whether they are correct or faulty.

Definition 2 (uniform interactive consistency). *A protocol solves* uniform interactive consistency (UIC) *if it satisfies the following properties:*

- *(UIC-Termination) Every correct process eventually decides.*
- *(UIC-Validity) The decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i is not faulty.*
- *(UIC-Uniform Agreement) No two different vectors are decided.*

Parvédy and Raynal [23] studied the problem of UIC in the context of general omission failures. They give an algorithm that solves UIC in such systems provided a majority of processes is correct. Since their system model is the same as the one used for trusted systems in this paper, we conclude:

Theorem 1 ([23]). *If $t < n/2$ then UIC is solvable in the trusted system.*

Parvédy and Raynal also show that Uniform Consensus (UC), a problem closely related to UIC, can be solved in the omission failure model only if $t < n/2$. In Uniform Consensus, instead of agreeing on a vector of input values like in UIC, all processes have to decide on a single value which must be input value of some process. Given a solution to UIC, the solution to UC can be constructed in the following way: the processes first solve UIC on their input values and then output the first non-$\perp$ element of the decided vector as the result of UC. Thus, we conclude:

Corollary 1. *UIC is solvable in the trusted system only if $t < n/2$.*

5 Maintaining Secrecy in Trusted Systems

The problem of UIC, which was introduced in the previous section, will be used as a building block in our solution to SMC. The idea is that a protocol for SMC will delegate certain security-critical actions to the trusted system in which the UIC protocol runs. In this section we argue that we have to carefully analyze the security properties of the protocols which run within the trusted system in order to maintain confidentiality and be able to implement SMC-Privacy.

5.1 Example of Unauthorized Information Flow

Assume that a host H_A hands its input value v_A of SMC to its associated security module M_A and that M_A sends v_A over a secure channel to the security module M_B of host H_B. Since all communication travels through H_B (see Fig. 2), H_B can derive some information about v_A even if all information is encrypted. For example, if no special care is taken, H_B could deduce the size (number of bits) of v_A by observing the size of the cipher text of v_A. This may be helpful to exclude certain choices of v_A and narrow down the possibilities in order to make a brute-force attack feasible.

As another example, suppose M_A only sends v_A to M_B if v_A (interpreted as a binary number) is even. Since we must assume that H_B knows the protocol which is executed on M_A and M_B, observing (or not observing) a message on the channel at the right time is enough for M_B to deduce the lowest order bit of v_A. In this example, the control flow of the algorithm (exhibited by the message pattern) unintentionally leaks information about secrets.

5.2 Security Properties of Protocols in the Trusted System

A protocol running in the trusted system needs to satisfy two properties to be of use as a building block in SMC:

- (Content Secrecy) Hosts cannot learn any useful information about other hosts' inputs from observing the *messages* in transit.
- (Control Flow Secrecy) Hosts cannot learn any useful information about other host's inputs from observing the *message pattern*.

To provide these two secrecy properties in general, it is sufficient to use a communication protocol between the processes that ensures *unobservability*. Unobservability refers to the situation when an adversary cannot distinguish meaningful protocol actions from "random noise" [26]. In particular, unobservability assumes that an adversary knows the protocol which is running in the underlying network. It demands that despite this knowledge and despite observing the messages and the message pattern on the network it is impossible for the adversary to figure out in what state the protocol is. The term "state" refers to all protocol variables including the program counter, e.g., the mere fact whether the protocol has started or has terminated must remain secret.

Definition 3 (unobservability). *A protocol satisfies* unobservability *if an unauthorized entity which knows the protocol cannot learn any information about the state of the protocol.*

Obviously, if unobservability is fulfilled during the application of UIC then an adversary cannot obtain any information which may be derived from the control flow of the algorithm and therefore Content Secrecy and Control Flow Secrecy are fulfilled.

There are known techniques in the area of unobservable communication that guarantee perfect unobservability [26]. It goes without saying that unobservability techniques are not for free. However, the cost does not depend on the function F and depends only polynomially on the number of processes (i.e. number of real messages).

Unobservability as well as Content and Control Flow Secrecy are sufficient to maintain secrecy in the trusted subsystem. However, Control Flow Secrecy is sometimes not necessary. In the fair exchange implementation of Avoine *et al.* [3] it was shown that, to ensure security, it is sufficient that the adversary does not know when the agreement protocol is in its *final* round. The adversary may know whether the protocol is running or not.

This (weaker) form of Control Flow Secrecy was implemented using the idea of *fake rounds*. In the first round of the protocol, a random number ρ is distributed among the security modules. Before actually executing the agreement protocol, ρ fake rounds are run. Using encryption and padding techniques, it is impossible for the adversary to distinguish a fake round from an actual round of the agreement protocol.

6 Solving SMC with Security Modules

6.1 Main Result

The following theorem shows that SMC and UIC are "equivalent" in their respective worlds. For lack of space, the proof can be found in Appendix A. The idea of the proof is to distribute the input values to the function F within the trusted subsystem using UIC and then evaluate F on the resulting vector. The functional properties of SMC correspond to those of UIC while the security properties of SMC are taken care of the properties of the security modules and the fact that the UIC protocol can be made to operate in an unobservable way.

Theorem 2. *SMC is solvable for any deterministic F in the untrusted system if and only if UIC is solvable in the associated trusted system.*

Theorem 2 allows us to derive a lower bound on the resilience of SMC in the given system model using Theorem 1.

Corollary 2. *There is no solution to SMC in the untrusted system if $t \geq n/2$.*

6.2 Analysis

Theorem 2 and Corollary 2 show that adding security modules cannot improve the resilience of SMC compared to the standard model without trusted hardware [16]. For the model of perfect security (i.e., systems without cryptography) our secure hardware has a potential to improve the resilience from a two-thirds majority [5, 7] to a simple majority. However, this would rely on the assumption that security modules can withstand *any* side-channel attack, an assumption which can hardly be made in practice.

Since F is computed locally, the main efficiency metric for our solution is message and round complexity of the underlying UIC protocol. For example, the worst case message complexity of the protocol of Parvédy and Raynal [23] is $O(n^3)$ and the worst case round complexity if $O(n)$ even if modifications for unobservable communication are added [6]. This is in contrast to the most efficient solution to SMC without secure hardware which requires at least $O(n^2)$ rounds and $O(mn^3)$ messages where m is the number of multiplications in F [18].

7 TrustedPals Implementation

We now report on the implementation of the trusted subsystem using smart cards [1]. Our implementation is fundamentally a peer-to-peer distributed system. Each peer consists of two parts, the security module and its (untrusted) host application. It leverages the power of different programming platforms and software technologies.

7.1 Programming Platforms and Technologies

The security module is realized using a Java Card Technology enabled smart card, whereas its (untrusted) host application is implemented as a normal Java desktop application.

The Java Card Technology defines a subset of the Java platform for smart cards and allows application developers to create and install smart card applications on their own, even after the smart card was manufactured. Multiple Java Card applications (so-called *Applets*) from different vendors can run on the same smart card, without compromising each other's security. The communication between the host computer and the smart card is a half-duplex, master-slave model. In Java Card Technology, there are two programming models used for communication with the smart card, the *APDU* message-passing model and the Java Card Remote Method Invocation (*JCRMI*), a subset of Java SE RMI distributed-object model. Though our current implementation uses the APDU model, we plan to migrate to JCRMI. For more information about the Java Card Technology, we refer the interested readers to Chen [8].

The host part of the application is implemented in Java, using the *Spring Framework* [13]. Spring is a lightweight, dependency injection inversion of control

container that allows us to easily configure and assemble the application components. For more on dependency injection and inversion of control containers, please refer to Fowler [12].

The *OpenCard Framework* [17] is used by the host part of the application for communication with the smart card, whereas the *JMS* (Java Message Service) is used for communication with the other hosts. JMS is a standard Java API, part of Java Platform, Enterprise Edition, for accessing enterprise messaging systems. It allows the applications to communicate in a loosely coupled, asynchronous way. Besides, the provided infrastructure supports different quality of service, fault tolerance, reliability, and security requirements. As a JMS provider, we have used *ActiveMQ* [2]. It is an open source, 100% compliant JMS 1.1 implementation, written in Java. ActiveMQ can be seamlessly integrated and configured through Spring. What is more, its support for *message-driven POJOs* concept and the Spring's JMS abstraction layer makes much easier the development of messaging applications. Another interesting feature of ActiveMQ is its concept called *Networks of Brokers*. It allows us to start a broker for each host; these brokers then interconnect with each other and form a cluster of brokers. Thus, a failure of any particular host does not affect the other hosts.

7.2 Architecture

The overall (simplified) architecture of our project is depicted in Figure 3, where relations between components are depicted. The components comprise the Protocol on the smart card, the Controller, the Clock, the Message Communication Unit and the Adversary which operate on the host.

The Protocol on the smart card in fact consists of several parts: (1) a Java implementation of the function F which is to be evaluated, (2) an implementation of UIC which satisfies sufficient secrecy properties as explained above, and (3) provisions to set up confidential and authenticated channels with other smart cards.

The idea is that all messages generated during the execution of the protocol are encrypted and padded to a standard length and sent over the APDU interface to the host. The Communication Unit takes care of distributing these messages to other hosts using standard Internet technology. The Controller and the Clock are under total control of the host. This is acceptable as tampering with them would just result in message deletion or in disturbing the synchronous time intervals, which might cause message losses as well. That would be no problem, since the adversary has power to destroy messages stored in the Message Communication Unit, due to the general omission model nature, where process crashes and message omissions may occur - note that a process crash may be simulated by permanent message omissions, that is, all messages are omitted. In order to perform fault-injection experiments, the Adversary unit may simulate transient or permanent message omissions.

In the host memory, the Clock triggers individual rounds in the Controller, which then triggers the individual rounds in the protocol on the smart card. To run a different protocol, it is sufficient to add a new (so called applet) protocol

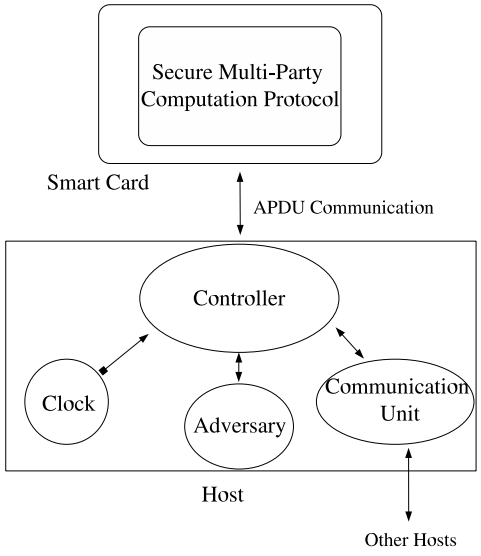

Fig. 3. Relations between architecture components.

to the (multiple applet) smart card. More information on the implementation can be found in Appendix B.

7.3 Experiences

We implemented the fair exchange protocol of Avoine *et al.* [3] within TrustedPals and tested it with four participating parties. The code of one party was executed on a smart card while the code of the other parties was simulated on a single PC.

We did some initial measurements on the speed of the protocol. We observed that the communication between the host PC and the smart card is the bottleneck and dominates the time needed to execute a synchronous round. Our implementation needed roughly 600 ms to perform the necessary communication and so in our setup we set the round length to one second. Depending in the necessary level of security the protocol needs between 4 and 10 rounds (i.e., between 4 and 10 seconds) to complete. Since our implementation is not optimized for speed and future technologies promise to increase the communication bandwidth on between host and smart card, we believe that this performance can be improved considerably.

We did some fault-injection experiments and tested the implementation using the random adversary. Within more than 10.000 runs the protocol did not yield any single successful security violation.

References

1. Trustedpals source code. Downloadable from `http://pi1.informatik.uni-mannheim.de`, April 2006.
2. ActiveMQ. http://activemq.codehaus.org.
3. Gildas Avoine, Felix Gärtner, Rachid Guerraoui, and Marko Vukolic. Gracefully degrading fair exchange with security modules. In *Proceedings of the Fifth European Dependable Computing Conference*, pages 55–71. Springer-Verlag, April 2005.
4. Gildas Avoine and Serge Vaudenay. Fair exchange with guardian angels. In *The 4th International Workshop on Information Security Applications – WISA 2003*, Jeju Island, Korea, August 2003.
5. Michael Ben-Or, Shafi Goldwasser, and Ari Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual Symposium on Theory of Computing (STOC)*, pages 1–10, Chicago, IL USA, May 1988. ACM Press.
6. Zinaida Benenson, Felix C. Gärtner, and Dogan Kesdogan. Secure multi-party computation with security modules. Technical Report AIB-10-2004, RWTH Aachen, December 2004.
7. David Chaum, Claude Crepeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In Richard Cole, editor, *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pages 11–19, Chicago, IL, May 1988. ACM Press.
8. Z. Chen. *Java Card Technology for Smart Cards - 1st Edition*. Addison-Wesley Professional, 2000.

9. Carole Delporte-Gallet, Hugues Fauconnier, and Felix 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, October 2005.
10. Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, and Steve Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.
11. E.Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software - 1st Edition*. Addison-Wesley Professional, 1995.
12. M. Fowler. *Inversion of Control Containers and the Dependency Injection Pattern*. http://martinfowler.com/articles/injection.html.
13. Spring Framework. http://www.springframework.org.
14. F. Freiling, M. Herlihy, and L. Penso. Optimal randomized fair exchange with secret shared coins. In *Proceedings of the Ninth International Conference on Principles of Distributed Systems*. Springer, December 2005.
15. Oded Goldreich. Secure multi-party computation. Internet: `http://www.wisdom.weizmann.ac.il/~oded/pp.html`, 2002.
16. Oded Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proceedings of the 19th ACM Symposium on the Theory of Computing (STOC)*, pages 218–229, 1987.
17. U. Hansmann, M. Nicklous, T. Schäck, A. Schneider, and F. Seliger. *Smart Card Application Development Using Java - 2nd Edition*. Springer, 2002.
18. Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *Proceedings of Asiacrypt*, 2000.
19. Alexander Iliev and Sean Smith. More efficient secure function evaluation using tiny trusted third parties. Technical Report TR2005-551, Dartmouth College, Computer Science, Hanover, NH, July 2005.
20. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
21. P. MacKenzie, A. Oprea, and M.K. Reiter. Automatic generation of two-party computations. In *SIGSAC: 10th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2003.
22. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — A secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.
23. Philippe Raïpin Parvédy and Michel Raynal. Uniform agreement despite process omission failures. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*. IEEE Computer Society, April 2003. Appears also as IRISA Technical Report Number PI-1490, November 2002.
24. M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
25. Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.
26. Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity — A proposal for terminology. In H. Federrath, editor, *Anonymity 2000*, number 2009 in Lecture Notes in Computer Science, pages 1–9, 2001.
27. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

28. Trusted Computing Group. Trusted computing group homepage. Internet: `https://www.trustedcomputinggroup.org/`, 2003.
29. A. C. Yao. Protocols for secure computations (extended abstract). In *23th Annual Symposium on Foundations of Computer Science (FOCS '82)*, pages 160–164, Los Alamitos, Ca., USA, November 1982. IEEE Computer Society Press.
30. Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. *TOCS*, 20(4):329–368, November 2002.

A Proof of Theorem 2

Statement: SMC is solvable for any deterministic F in the untrusted system if and only if UIC is solvable in the associated trusted system.

Proof. ($\Leftarrow$) We first prove that the solvability of UIC in the trusted system implies the solvability of SMC in the untrusted system. Fig. 4 shows the transformation protocol which is executed within the security module. The hosts first give their inputs for SMC to their security modules. Security modules run "secure UIC" (i.e., UIC which satisfies Message Secrecy and Control Flow Secrecy) on these inputs, compute F on the decided vector and give the result to their hosts. We prove that the properties of SMC are achieved.

First consider *SMC-Validity*. UIC-Termination guarantees that all correct processes eventually decide on some vector D. UIC-Validity guarantees that D contains the inputs of all correct processes, and hence, SMC-Validity holds for the output of the transformation algorithm.

Consider *SMC-Agreement*. From UIC-Uniform Agreement it follows that all processes decide on the same vector. As F is deterministic, all processes compute the same F-result if they compute such a result.

SMC-Termination follows immediately from UIC-Termination.

Now consider *SMC-Privacy*. Since secure UIC is executed (i.e., the construction and proof techniques presented in Section 5 have been applied to ensure Content Secrecy and Control Flow Secrecy) and because security modules are tamper-proof, we conclude that there is no unauthorized information flow from within the trusted system to the outside (i.e., to the untrusted system). The only (authorized) flow of information occurs at the interface of the security modules when they output the result of computing F. SMC-Privacy easily follows from this observation.

($\Rightarrow$) We now prove that if SMC is solvable in the untrusted system, then UIC is solvable in the trusted system.

First note that if SMC is solvable in the untrusted system, then SMC is trivially also solvable in the trusted system. This is because the assumptions available to the protocol are much stronger (general omission failures instead of Byzantine).

To solve UIC, we let the processes compute the function $F(v_1, \ldots, v_n) = (v_1, \ldots, v_n)$ (see Fig. 5). We now show that the properties of UIC follow from the properties of SMC.

UIC-Termination follows immediately from SMC-Termination.

To see *UIC-Validity*, consider the decided vector $D = (d_1, \ldots, d_n)$. SMC-Validity and the construction of Fig. 5 guarantee that D contains the inputs of all correct processes. Consider a faulty process p_j with input value v_j. Then either F was computed using its input, and then $d_j = v_j$, or, according to our definition of SMC, function F was computed using a special input value $\perp$ instead of v_j, and then, $d_j = \perp$.

To see *UIC-Uniform Agreement* follows directly from SMC-Agreement.

This concludes the proof. □

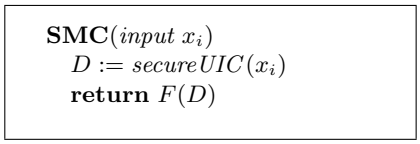

Fig. 4. Implementing SMC using UIC on security modules. Code for the security module of host H_i. The term "secure UIC" refers to a UIC protocol that satisfies Content Secrecy and Control Flow Secrecy.

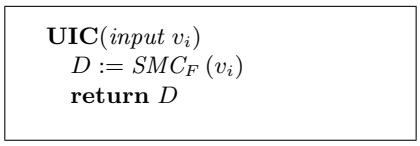

Fig. 5. Implementing UIC on security modules using SMC for the function $F(v_1, \ldots, v_n) = (v_1, \ldots, v_n)$ in the untrusted system. Code for the security module of host H_i.

B Details on Architecture and Operation of TrustedPals

B.1 Architecture

In Figure 6 more detailed information on the architecture of the components, seen as Java classes, can be found. The central piece of the architecture is the `Controller` class. It has associations to `MessageSender` and `MessageReceiver` classes, that, as their names imply, are responsible for handling the sending and receiving of messages, respectively.

To simulate sending and receiving omissions, the controller has associated two adversary strategies. We have defined different adversary behaviors: the `HonestAdversary` allows all messages to be sent out/received, whereas the `NastyAdversary` drops all messages. Other adversaries represent random behavior, etc.

Finally, the controller has a reference to a `Protocol` interface implementation (protocol) as in Figure 9, that represents the actual algorithm, that the system is performing. To date, we implemented one application function F, the gracefully degrading fair exchange functionality [3], and two UIC protocols, the adapted protocol of Parvédy and Raynal [23] and the ConsensusS protocol by Freiling, Herlihy and Penso [14]. More can be added easily. For each of these protocols, we have a smart card/Java Card based version and a pure Java implementation. The pure Java version serves us mainly as a proof of concept during the development and allows us to test the whole distributed system even on one computer with only one smart card reader. The Java Card implementation of each protocol consists then of two parts. One is the actual Java Card applet, that is deployed on the smart card and the host side implementation of the `Protocol` interface (always prefixed with "SmartCard"), that communicates with this applet.

The difference between the pure Java implementations of the protocols and the Java Card applet versions is also in the way they are configured. Whereas a simple *.properties* configuration file is used to supply the necessary configuration values to the Java versions, a more sophisticated GUI application was developed for entering configuration data and sending them to the smart card.

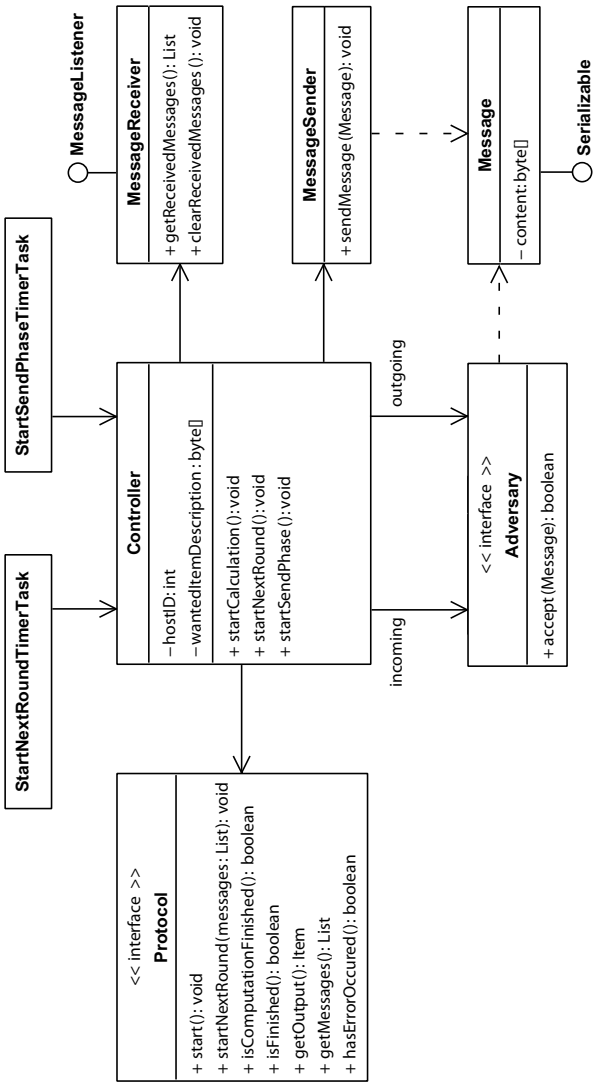

Fig. 6. Architecture.

Using the Spring Framework's powerful inversion of control dependency injection container, we are able to easily configure the controller, which protocol to use for the current application run, without the need to recompile the code. The same holds for the choice of current incoming and outgoing adversary strategies as well[1].

The main purpose of the `Message` class is to parse the byte array content, that is retrieved from the smart card and interpret it in a human readable form. This way, we can simply log the string representation of every message into database, and thus easily follow what is going on in the system. To send the message, we simply use a JMS `ObjectMessage` to encapsulate and transfer the `Message` instance to its destination.

B.2 Operation

The system operation proceeds in rounds. We have divided each round into two phases, the computation phase and the sending phase. Proper synchronization between different hosts is achieved by configuring all the hosts to use the same round length (i.e. the same computation phase length and the same sending phase length), and by starting them all at about the same time. Right after the start, the controller calls the `start()` method on the associated protocol to start the computation.

There are two timer threads responsible for the life cycle of each host module, that are represented by classes `StartNextRoundTimerTask` and `StartSendPhaseTimerTask` in our diagram. Their functionality, outside configuration and smooth integration is achieved through Spring Framework's support for job scheduling, which we won't discuss here in detail. As a rough idea, the reader can imagine that there are two separate threads with associated time intervals, that are sleeping most of the time. Periodically, at the given time interval, each of this threads wakes up and notifies the controller by calling its methods `startNextRound()` and `startSendPhase()`, respectively. These methods then implement the lifecycle of the system by invoking the respective methods on the current protocol, that we discuss below.

Note that every method call in the `Protocol` implementation class, that involves communication with the smart card, is invoked from within a separate thread to prevent it to block the timer threads. In the sending phase (see Fig. 7), the controller first verifies, if no error occurred during the computation (line 1). If not, it checks, whether the computation for the current round has finished successfully (line 5). If so, it checks if the whole protocol has finished (line 9). In this case, it retrieves the item and verifies, if it is actually the correct one (line 11). In case the whole protocol is not finished yet, but the computation for the current round finished successfully (line 13), it retrieves the messages to be sent out during this round (line 14) and sends them using `MessageSender` (in case they are not dropped by the outgoing adversary (line 18)). At the same time, the `MessageReceiver` asynchronously listens to its associated JMS topic and collects all received messages.

In the computation phase (see Fig. 8), the controller then retrieves the received messages from `MessageReceiver` (line 1), filters them through the incoming adversary (line 3) and passes the rest to the `startNextRound()` method of the protocol to start the next round (line 9). This process repeats until the protocol finishes successfully or until an error occurs.

[1] This design possibility of interchangeable family of algorithms is referred to as the Strategy design pattern [11].

```
01:    if (protocol.hasErrorOccured()) {
02:        terminate application with error
03:    }
04:
05:    if (not protocol.isComputationFinished()) {
06:        terminate application with error
07:    }
08:
09:    if (protocol.isFinished()) {
10:        receivedItem = protocol.getItem()
11:        verify receivedItem
12:
13:    } else {
14:        messages = protocol.getMessages()
15:
16:        foreach message in messages {
17:            if (outgoingAdversary.accept(message)) {
18:                messageSender.sendMessage(message)
19:            }
20:        }
21:    }
```

Fig. 7. Pseudocode of the sending phase.

```
01:    messages = messageReceiver.getReceivedMessages()
02:
03:    foreach message in messages {
04:        if (not incomingAdversary.accept(message)) {
05:            drop message from messages
06:        }
07:    }
08:
09:    protocol.startNextRound(messages)
10:    messageReceiver.clearReceivedMessages()
```

Fig. 8. Pseudocode of the computation phase.

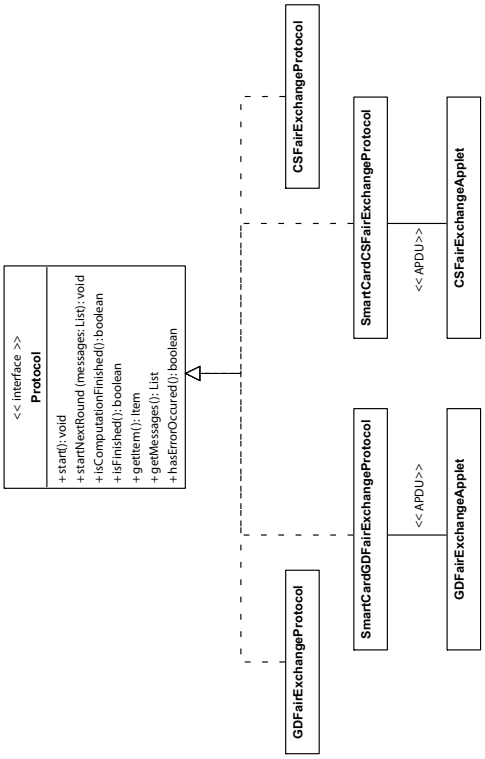

Fig. 9. Protocol hierarchy.