Optimal Randomized Fair Exchange with Secret Shared Coins

Felix Freiling[1], Maurice Herlihy[2⋆], and Lucia Draque Penso[3⋆⋆]

[1] Computer Science Department, University of Mannheim, D-68131 Mannheim, Germany
[2] Computer Science Department, Box 1910, Brown University, Providence, RI 02912, U.S.A.
[3] Computer Science Department, RWTH Aachen University, D-52056 Aachen, Germany

Abstract. In the *fair exchange* problem, mutually untrusting parties must securely exchange digital goods. A fair exchange protocol must ensure that no combination of cheating or failures will result in some goods being delivered but not others, and that all goods will be delivered in the absence of cheating and failures.

This paper proposes two novel randomized protocols for solving fair exchange using simple trusted units. Both protocols have an optimal expected running time, completing in a constant (3) expected number of rounds. They also have optimal resilience. The first one tolerates any number of dishonest parties, as long as one is honest, while the second one, which assumes more aggressive cheating and failures assumptions, tolerates up to a minority of dishonest parties.

The key insight is similar to the idea underlying the *code-division multiple access* (CDMA) communication protocol: outwitting an adversary is much easier if participants share a common, secret pseudo-random number generator.

1 Introduction

In the *fair exchange* problem, a set of parties want to trade an item which they have for an item of another party (for a survey of fair exchange see [11]). Fair exchange is a fundamental problem in domains with electronic business transactions since (1) items can be any type of electronic asset (electronic money, documents, music files, etc.) and (2) fairness is especially important in rather anonymous environments without means to establish mutual trust relationships. Briefly spoken, fair exchange guarantees that (1) every honest party eventually

⋆ Maurice Herlihy was supported by Deutsche Forschungsgemeinschaft (DFG) when visiting RWTH Aachen University.

⋆⋆ Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg "Software for Mobile Communication Systems" at RWTH Aachen University.

either delivers its desired item or aborts the exchange, (2) the exchange is successful if no party misbehaves and all items match their descriptions, and (3) the exchange should be fair, i.e., if the desired item of any party does not match its description, then no party can obtain any (useful) information about any other item. Fair exchange algorithms must guarantee these properties even in the presence of arbitrary (malicious) misbehavior of a subset of participants.

Fair exchange, a security problem, can be reduced [2] to a fault-tolerance problem, namely a special form of *uniform consensus*. In the (non-uniform) consensus problem [13], each process in a group starts with a private input value, and after some communication, each non-faulty process is required to decide (termination) on the same private output value (agreement), so that all processes that decide choose some process's private input value (validity). In its uniform version, however, agreement requires all processes that decide (faulty or non-faulty) to decide the same value. Only non-faulty processes are required to terminate.

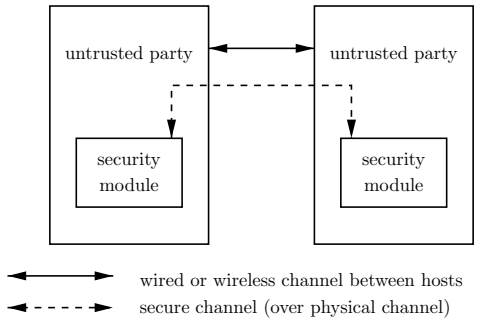

Fig. 1. Untrusted parties and security modules.

The reduction from fair exchange to consensus [2] holds in a synchronous model where each participating party is equipped with a trusted unit, that is, a tamper-proof security module like a smart card (see Fig. 1). Security modules have recently been advocated by key players in industry to improve the security of computers in the context of *trusted computing* [15]. Today, products exist which implement such trusted devices (see for example [7]). Roughly speaking, a security module is a certified piece of hardware executing a well-known algorithm. Security modules can establish confidential and authenticated channels between each other. However, since they can only communicate by exchanging messages through their (untrusted) host parties, messages may be intercepted or dropped. Overall, the security modules form a *trusted subsystem* within the overall (untrusted) system. The integrity and confidentiality of the algorithm running in the trusted subsystem is protected by the shield of tamper proof hardware. The integrity and confidentiality of data sent across the network is protected by standard cryptographic protocols. These mechanisms reduce the type and nature of adversarial behavior in the trusted subsystem to message

loss and process self-destruction, two standard fault-assumptions known under the names of *omission* and *crash* in the area of fault-tolerance.

This paper proposes two novel randomized protocols for solving uniform consensus with binary inputs (and hence fair exchange) using such trusted units. Our protocols are time optimal, completing in a constant (3) expected number of rounds. They are also optimal in terms of resilience. The key insight is similar to the idea underlying the *code-division multiple access* (CDMA) communication protocol [16]: outwitting an adversary is much easier if participants share a common, secret pseudo-random number generator. In a multi-round protocol, each trusted unit can flip a coin, and take action secure in the knowledge that every other trusted unit has flipped the same value, and is taking a compatible action in that round. Because messages are encrypted, coin flip outcomes can be hidden, so dishonest parties can neither observe past coin flips nor predict future ones. (Of course, the pseudo-random algorithm itself need not be secret as long as the trusted units' common seed is kept secret, just like their common cryptographic key.) We believe that this approach is both efficient and practical.

The presentation is structured as follows. In section 2 we describe the model of computation considered, whereas in section 3 we show how to reduce fair exchange to uniform consensus. Section 4 displays related work. Optimal randomized uniform consensus protocols for binary inputs with a constant (3) expected number of rounds are introduced in sections 5 and 6. Note that both protocols may be generalized to a larger set of k values with an extra factor cost of $\log(k)$. However, we concentrate on the binary case, since we are mainly interested in solving fair exchange efficiently. Finally, we conclude with section 7, where a summary and work future directions are exhibited.

2 Model of Computation

Our model of computation is essentially synchronous: participants exchange messages in synchronous rounds. Of course, real distributed systems are not synchronous in the classical sense, but it is reasonable to assume an upper bound on how long one can expect a non-faulty processor to take before responding to a message. A processor that takes too long to join in a round is assumed to be faulty or malicious.

The system is logically structured into an untrusted system (including the untrusted parties and their communication channels) and the trusted subsystem consisting of the parties' individual trusted units, that is, their tamper-proof security modules (see Fig. 2). The untrusted parties can interact with their trusted units through a well-defined interface, but they cannot in any other form influence the computation within the trusted unit.

As noted, communication among the trusted units is confidential and authenticated, so malicious parties cannot interpret or tamper with these messages. Because each trusted unit sends the same encrypted message to every other trusted unit, we have receiver anonymity and so a cheating party cannot learn who is communicating to who from traffic analysis. An untrusted party can, however,

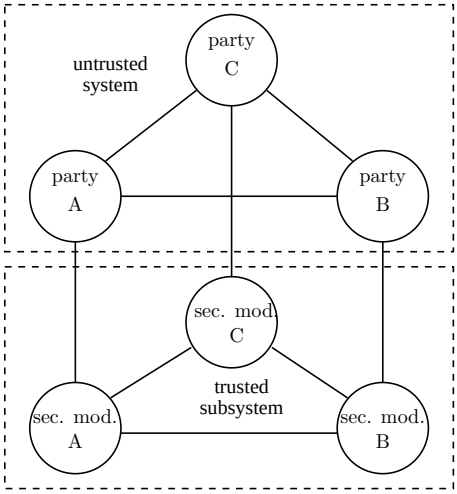

Fig. 2. The untrusted system and the trusted subsystem.

prevent outgoing messages from being sent (called a *send omission*), or incoming messages from being received (called a *receive omission*) or destroy its trusted unit (called a *crash*). The effects of a crash can be regarded as a permanent send (and receive) omission.

Define a party as *cheating* if it causes send or receive omissions of its trusted unit. A party which does not cheat is *honest*. A fair exchange protocol must ensure that under no circumstances will goods be delivered to a cheating party but not to all honest parties. It is, however, acceptable to deliver the goods to all honest parties, but not to some cheating parties. Cheating may cause the exchange to *fail*, so that no goods are delivered to any party. In the absence of cheating, the exchange should *succeed*, causing goods to be delivered to all participants. For brevity, we refer to processes when we really mean untrusted processes equipped with trusted units. With a *process failure* we mean either a crash, a send message omission or a receive message omission.

3 Fair Exchange as Consensus

The reduction from fair exchange to uniform consensus works as follows. In the first round of the protocol, each party applies its acceptance test to the encrypted digital goods received from the others (in special cases this test can also be performed within the trusted unit). It then informs its trusted unit whether the goods passed the test. The trusted units broadcast this choice (using confidential and authenticated messages) within the trusted subsystem. Each unit that receives unanimous approvals starts the consensus protocol with input 1, and each trusted unit that either observes a disapproval or no message from a trusted unit starts the consensus protocol with input 0. At the end of the

protocol, each trusted unit delivers the goods if the outcome of the protocol is 1, and refuses to do so if the outcome is 0.

It is easy to see that in the absence of failures or cheating all goods will be delivered. The uniform consensus protocol ensures that all honest parties agree on whether to deliver the goods, and its uniformity ensures that no trusted unit residing at a cheating party will deliver the goods if any trusted unit at an honest party decides not to. (Recall that it is acceptable if the honest processes deliver the goods after deciding 1, even if a cheating process fails to deliver the goods after deciding 0.)

As noted, the protocols considered in this paper are *randomized*, in the sense that they rely on the assumption that trusted units generate pseudo-random values that cannot be predicted by an adversary. These protocols always produce correct results, but their running time is a random variable, the so-called *Las Vegas* model. (It is straightforward to transform these protocols into *Monte-Carlo* protocols that run for a fixed number of rounds, and produce correct results with very high probability.)

To simplify the presentation, we first present an uniform consensus protocol that works in a failure model that permits send, but not receive omissions. This protocol is slightly simpler and more robust: it tolerates $f < n$ cheating processes, while the full send/receive omissions model protocol tolerates $f < n/2$ failures. Both resilience levels are optimal for their respective models [12]. Presenting the protocol in two stages illustrates how assumptions about the model affect the protocol's complexity and resilience.

4 Related Work

We build on work of Parvédy and Raynal [12]. They derive optimal early stopping deterministic uniform consensus algorithms for synchronous systems with send or send/receive omission failures. However, our algorithms are more efficient in most cases (if the number of failures is not constant) and at least comparable (otherwise).

Feldman and Micali [8] exhibit optimal consensus algorithms for Byzantine agreement, which in principle could also be used in omission failure models. Despite having also an optimal expected running time, our algorithms outperform theirs both on resilience and on the probability of not having termination violated.

Avoine, Gärtner, Guerraoui and Vukolic [2] show how to reduce the fair exchange problem in a system where processes are provided with security modules to the consensus problem in omission failure models. A solution to the fair exchange problem is presented by use of the algorithms of Parvédy and Raynal [12]. In the same context, Delporte, Fauconnier and Freiling [6] investigate solutions to consensus for *asynchronous* systems which are equipped with unreliable failure detectors. They exhibit a weak failure detector in the spirit of previous work by Chandra, Hadzilacos and Toueg [4] that allows to solve asynchronous consensus in omission failure environments.

Aspnes [1] presents a survey of randomized consensus algorithms for the shared memory model where processes are prone to crashes. These results are particularly interesting, since consensus cannot be solved deterministically in a pure asynchronous distributed system, as proved in [9] by Fischer, Lynch and Paterson.

Finally, Chaudhuri [5] introduces the k-set agreement problem, a generalization of the consensus problem, and proves it to be harder. Later, Borowsky and Gafni [3], Herlihy and Shavit [10], and Saks and Zaharoglou [14] would demonstrate that there is no wait-free protocol for k-set agreement (or consensus) in asynchronous message-passing or read/write memory models.

5 Optimal Protocol for Send Omissions

The *ConsensusS* algorithm in Figure 3 solves uniform consensus with binary inputs in optimal 3 expected synchronous rounds tolerating an optimal number of up to $f < n$ failures - send message omissions as well as process crashes.

As noted, all processes share a common secret seed and pseudo-random number generator. We denote the r-th such pseudo-random binary number by *flip*(r). For each r, every process computes the same value for *flip*(r).

```
1  send prefer(binary preference) to all;
2  for each round r {
3    if (both prefer(0) and prefer(1) received) {
4      send disagreement(r) to all;
5    }
6    on (receipt of disagreement(r) for the first time) {
7      send disagreement(r) to all;
8    }
9    if (all received preferences are prefer(v)) and (no disagreement(r) received){
10     if (flip(r) == v) {
11       send decide(v) to all and return(v);
12     } else {
13       send prefer(v) to all;
14     }
15   } else {
16     send prefer(flip(r)) to all;
17   }
18   if (any decide(v) received) {
19     return(v)
20   }
21 }
```

Fig. 3. Uniform consensus for send message omissions and process crashes.

In ConsensusS, each process broadcasts its binary input (line 1). In each subsequent round, the process waits to hear each process's *preference*. If they disagree (line 3), the process broadcasts a message informing the others. When receiving such a broadcast for the first time (line 6), every process relays it. Hence, if any non-faulty process receives mixed preferences or a $disagreement(r)$ message, then all processes receive a $disagreement(r)$ message and will change preference according to the coin flip. If they agree (line 9) and no message communicating disagreement seen by another process is received, then the process checks whether that preference agrees with the common pseudo-random binary number for that round. If so, it is safe to decide that value (line 11). If not, the process simply rebroadcasts the preference (line 13). If the preferences disagree or the process is informed so, then the process uses the common pseudo-random binary number to choose a new preference (line 16). If any process announces that it has decided, then the process decides on the same value (line 18).

Very informally, this protocol exploits in an essential way the observation that each process (but not the adversary) can predict the others' next coin flips. If a process receives v from all processes, then v was sent by at least one good process, so every other process will either receive all v preferences or both preferences. Any processes that receive either mixed preferences or $disagreement(r)$ messages will change preference according to the coin flip. If the coin flip is the same as v, then all processes will prefer v, and it is safe to decide.

Lemma 1. *If $f < n$, for every process the expected number of rounds of ConsensusS is 3, and the protocol terminates with probability 1.*

Proof. Think of an execution as a tree, where the root node represents the initial round and the children of a node represent the following round possibilities. Let $E(n)$ be the expected number of rounds from node n. If n has children $n.1$ and $n.2$, chosen by coin flip, then $E(n) = (1/2)(1 + E(n.1)) + (1/2)(1 + E(n.2))$. Each child contributes one plus its expected running time, but with probability one-half. Now let

- $E(n) = E_1(n)$ if at node n some non-faulty processes sent $prefer(0)$ and some non-faulty processes sent $prefer(1)$,
- $E(n) = E_2(n)$ if at node n all non-faulty processes sent $prefer(v)$ and some non-faulty processes receive a disagreement message or both $prefer(0)$ and $prefer(1)$,
- $E(n) = E_3(n)$ if at node n all non-faulty processes sent $prefer(v)$ and all non-faulty processes receive no disagreement messages and only $prefer(v)$.

Note that if $E(n) = E_z(n)$ and $E(n.1) = E_w(n.1)$, it may be that $z \neq w$. However, it is always the case that if $E(n.1) = E_z(n.1)$ then $E(n.2) = E_z(n.2)$. The reason is that from one round to the other the values that the non-faulty processes send and receive may change. However, if the non-faulty processes behave in a way at one children, then they should behave the same way at the other, since both children just differ in the coin flip. Hence, executions differing

themselves by the values sent and received by non-faulty processes may generate distinct execution trees.

Now let e be the root of an execution tree. Consider that

- $E(e) = E_1(e)$: If there are non-faulty processes that sent *prefer*(0) and other non-faulty processes that sent *prefer*(1) in round r, then at round $r+1$ every process receives at least one message *prefer*(0) and one message *prefer*(1), and thus, from round $r+1$ on, all preference messages sent by every process (and all received as well) will be *prefer*(*flip*$(r+1)$). Hence, all processes will decide on *flip*$(r+1)$ in the first round t such that *flip*$(t) = flip(r+1)$, and the probability that any process (and thus, a non-faulty one) violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that *flip*$(t) = flip(r+1)$ is 2. Thus, the expected number of rounds of ConsensusS is $3 = E_1(e) = (1/2)(1+2) + (1/2)(1+2)$.
- $E(e) = E_2(e)$: If all non-faulty processes sent *prefer*(v) in round r and part of the non-faulty processes receive a disagreement message or both messages *prefer*(0) and *prefer*(1) in round $r+1$, then all processes receive disagreement messages and from round $r+1$ on, all preference messages sent by every process (and all received as well) will be *prefer*(*flip*$(r+1)$). Thus, all processes will decide on *flip*$(r+1)$ in the first round t such that *flip*$(t) = flip(r+1)$, and the probability that any process (and thus, a non-faulty one) violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that *flip*$(t) = flip(r+1)$ is 2. Thus, the expected number of rounds of ConsensusS is $3 = E_2(e) = (1/2)(1+2) + (1/2)(1+2)$.
- $E(e) = E_3(e)$: If all non-faulty processes sent *prefer*(v) and receive no disagreement messages and only *prefer*(v) in round $r+1$, then if *flip*$(r+1) = v$, all non-faulty processes send *decide*(v) messages and then decide by returning v themselves. Moreover, on receipt of *decide*(v), all remaining processes decide by returning v. If *flip*$(r) \neq v$, then we fall again into the case that all non-faulty processes send *prefer*(v). That is, $E_3(e.2) = E_2(e.2) or E_3(e.2)$. Thus, the probability that any process (and thus, a non-faulty one) violates termination is zero and the expected number of rounds of ConsensusS is $3 = E_3(e) = (1/2)(1+1) + (1/2)(1+3)$.

In short, in all cases, if $f < n$, the probability that any process (and thus, a non-faulty one) violates termination is zero. Moreover, the expected number of rounds of ConsensusS is 3 for all processes. □

Lemma 2. *If $f < n$, each decided value is some process's input.*

Proof. Any decided value v is either an original input or the result of a shared coin flip. Consider the first *prefer*(*flip*(`r`)) statement to be executed, if any. In this case, there must have been a process received both *prefer*(0) and a *prefer*(1) messages, which means that some process had input value 0 and another had input value 1. It follows that either value is some process's input. □

Lemma 3. *If $f < n$, no two processes decide differently.*

Proof. Consider the first round r in which a process decides v. It must be the case that at round r, $\mathit{flip}(r) = v$ and all preference messages received by the process are $\mathit{prefer}(v)$. As the messages from all non-faulty processes are received by all processes and there is at least one non-faulty process, all processes receive at least one $\mathit{prefer}(v)$ message, and either decide on v at the same round r or send $\mathit{prefer}(v) = \mathit{prefer}(\mathit{flip}(r))$. It follows that from the next round $r + 1$ on, all messages sent from all processes (and thus, also all received ones) will be $\mathit{prefer}(v)$. Henceforth, no process can decide a value different from v. □

Theorem 1. ConsensusS *solves uniform consensus with binary inputs in a synchronous system prone to crashes and send message omissions, with a probability zero of termination violation, and both an optimal constant (3) expected rounds and an optimal $n - 1$ resilience (that is, up to $n - 1$ processes may be faulty: $f < n$).*

Proof. Follows directly from Lemmas 1, 2 and 3. □

6 Optimal Protocol for Send and Receive Omissions

The *ConsensusSR* algorithm in Figure 4 solves uniform consensus with binary inputs in optimal 3 expected synchronous rounds tolerating an optimal number of up to $f < n/2$ failures - send message omissions and receive message omissions as well as process crashes.

In ConsensusSR, all processes start by broadcasting their inputs (line 2). Whenever one process does not receive a message from another, it decides that process must be faulty, and ignores it from that point on (line 6). Even so, all non-faulty processes send and receive messages from one another. Moreover, a live faulty process always receives messages from at least one non-faulty process, since otherwise, it would have less than $n/2+1$ messages and it would halt before reaching a decision (line 7).

On each round, every process checks if all received messages contain the same preferred value v (line 9). If so, it broadcasts a message that it wants to decide on v (line 10). When receiving this message for the first time (line 12), processes relay it. If a process receives such message from a majority of processes (line 15) or if it receives a message to decide on v (line 18), then it sends messages to all processes to decide on v and returns v. Note that if a non-faulty process relays the message, all non-faulty processes will relay the message as well, so all non-faulty processes will receive the message from a majority of processes. As every process needs a non-faulty process to relay the message in order to decide on v, if any process decides on v, then every non-faulty process does as well. If a decision is not reached, then the process either sends a message with v as its current preference (line 22), if it received a majority of preferences v, or sends a message containing $\mathit{flip}(r)$ (line 24), otherwise.

```
1  Recipients = set of all processes;
2  send prefer(binary preference) to all;
3  foreach round r {
4    Received(r) = set of processes from which messages were received in round r
5    Recipients = Recipients intersection Received(r);
6    Messages(r) = set of messages received in round r which were sent by Recipients;
7    if (|Messages(r)|<n/2+1) {
8      halt; // too many failures
9      if (all in Messages(r) are prefer(v)) {
10       send want_decide(v) to Recipients;
11     }
12     on (receipt of want_decide(r,v) for the first time) {
13       send want_decide(r,v) to Recipients;
14     }
15     if (want_decide(r,v) received from majority of processes) {
16       send decide(v) to all and return(v);
17     }
18     on (receipt of decide(v)) {
19       send decide(v) to all and return(v);
20     }
21     if (majority in Messages(r) are prefer(v)) {
22       send prefer(v);
23     } else {
24       send prefer(\ flip (r));
25     }
26 }
```

Fig. 4. Uniform consensus for send and receive message omissions and process crashes.

Lemma 4. *On any single round after initialization (sending the binary private input), only one value is preferred or chosen deterministically.*

Proof. A process prefers or decides v deterministically only if it sees a majority for v.

□

Lemma 5. *If $f < n/2$, for every process the expected number of rounds of ConsensusSR is 3, and the protocol terminates with probability 1.*

Proof. After initialization (sending the binary private input), if all live processes send *prefer*(*flip*(r)) or if all live processes send *prefer*(v), they agree right away, by Lemma 4. If some send *prefer*(v) and some send *prefer*(*flip*(r)), again by Lemma 4, then all live processes will agree in the first round t such that $v = flip(r)$, and the probability that any non-faulty process violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that $v = flip(r)$ is 2.

Once agreement by all live processes is achieved, non-faulty processes will receive a majority of $want_decide(r, v)$, send decide(v) and return(v), immediately in the same round. This is because they always receive messages from each other, that is, they always belong to the *Recipients* of non-faulty processes, so once a non-faulty process sends a $want_decide(r, v)$ message, all non-faulty processes will send $want_decide(r, v)$ messages to (and receive them from) all non-faulty processes and guarantee a majority of $want_decide(r, v)$.

In short, in all cases, if $f < n/2$, the probability that a non-faulty process violates termination is zero. Moreover, the expected number of rounds of ConsensusSR is 3 for all processes.

□

Lemma 6. *If $f < n/2$, all processes in ConsensusSR decide some process's input.*

Proof. A decided value v, from $decide(v)$, is just obtained from a $prefer(v)$. Now, by induction, a v from $prefer(v)$ has to be either an input or a $flip(r)$ for some r. However, take the first $prefer(flip(r))$ to occur, if any do. In this case, a process received both a $prefer(0)$ and a $prefer(1)$, which means that there should be a proposed input value equal to 0 and another equal to 1, as the particular $prefer(flip(r))$ was the first one to take place. Otherwise, either there would be a majority of $prefer(v)$ or Hence, $flip(r)$ must be a proposed input value if any $prefer(flip(r))$ occurs, and v must also be one of the proposed values.

□

Lemma 7. *If $f < n/2$, agreement is never violated in ConsensusSR: no two processes decide differently.*

Proof. Consider the first round r when a process decides by returning v. Then, it must be the case that a majority of $want_decide(r, v)$ is received by the process. However, because each process deciding has to receive a $want_decide(r, v)$ from a non-faulty process and non-faulty processes always receive messages from each other, when any process has a majority of $want_decide(r, v)$, it must be the case that all non-faulty processes have a majority of $want_decide(r, v)$, that is, all non-faulty processes decide by returning v as well.

□

Theorem 2. *ConsensusSR solves uniform consensus with binary inputs in a synchronous system prone to crashes, send message omissions and receive message omissions, with a probability zero of termination violation, and both an optimal constant (3) expected number of rounds and an optimal $n/2 - 1$ resilience (that is, up to $n/2 - 1$ processes may be faulty: $f < n/2$).*

Proof. Follows directly from Lemma 5, 6 and 7.

□

7 Conclusions

The key idea in this paper is that if secure coprocessors can share secret cryptographic keys (as they do), then they can also share secret seeds for secure pseudo-random number generators. Such shared coins enable randomized (Las Vegas) algorithms for fair exchange and uniform consensus that are optimal in terms of expected running time and resilience.

Both the ConsensusS and ConsensusSR binary consensus protocols can be extended to a larger set of k values in $3\log(k)$ rounds via bit-by-bit consensus. It is an open question whether faster protocols exist (perhaps by doing bit-by-bit consensus in parallel).

References

1. James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3):165–175, September 2003.
2. 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.
3. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.
4. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J.ACM*, 43(4):685–722, July 1996.
5. Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 311–234. ACM Press, August 1990.
6. 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.
7. 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.
8. Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 148–161. ACM Press, May 1988.
9. Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.
10. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM (JACM)*, 46(6):858–923, November 1999.
11. Henning Pagnia, Holger Vogt, and Felix C. Gärtner. Fair exchange. *The Computer Journal*, 46(1), 2003.
12. Philippe Raïpin Parvédy and Michel Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 302–310. ACM Press, June 2004.

13. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreements in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.
14. Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, March 2000.
15. Trusted Computing Group. Trusted computing group homepage. Internet: `https://www.trustedcomputinggroup.org/`, 2003.
16. Andrew J. Viterbi. *CDMA : Principles of Spread Spectrum Communication*. Prentice Hall, 1995. ISBN 0201633744.