Secret Swarm Unit
Reactive $k-$Secret Sharing*
(Extended Abstract)

Shlomi Dolev[1], Limor Lahiani[1], and Moti Yung[2]

[1] Department of Computer Science,
Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{dolev,lahiani}@cs.bgu.ac.il
[2] Department of Computer Science,
Columbia University, New York, and Google, USA
moti@cs.columbia.edu

Abstract. Secret sharing is a basic fundamental cryptographic task. Motivated by the virtual automata abstraction and swarm computing, we investigate an extension of the k-secret sharing scheme, in which the secret components are changed on the fly, independently and without (internal) communication, as a reaction to a global external trigger. The changes are made while maintaining the requirement that k or more secret shares may reveal the secret and no $k-1$ or fewer reveal the secret.

The application considered is a swarm of mobile processes, each maintaining a share of the secret which may change according to common outside inputs e.g., inputs received by sensors attached to the process.

The proposed schemes support addition and removal of processes from the swarm as well as corruption of a small portion of the processes in the swarm.

Keywords: secret sharing, mobile computing.

1 Introduction

Secret sharing is a basic and fundamental technique [13]. Motivated by the high level of interest in the virtual automata abstraction and swarm computing, e.g., [3,2,1,4,5] we investigate an extension of the k-secret sharing scheme, in which the secret shares are changed on the fly, while maintaining the requirement that k or more shares reveal the secret and no $k-1$ or fewer reveal the secret.

There is a great interest in pervasive ad hoc and swarm computing [14], and in particular in swarming unmanned aerial vehicles (UAV) [9,4]. A unit of UAVs that collaborate in a mission is more robust than a single UAV that has to complete a mission by itself. This is a known phenomenon in distributed computing where a single point of failure has to be avoided. Replicated memory and state

* Partially supported by the Israeli Ministry of Science, Lynne and William Frankel Center for Computer Sciences and the Rita Altura trust chair in Computer Sciences.

K. Srinathan, C. Pandu Rangan, M. Yung (Eds.): Indocrypt 2007, LNCS 4859, pp. 123–137, 2007.

machine abstractions are used as general techniques for capturing the strength of distributed systems in tolerating faults and dynamic changes.

In this work we integrate cryptographic concerns into these abstractions. In particular, we are interested in scenarios in which some of the participants of the swarm are compromised and their secret shares are revealed. We would like the participants to execute a global transition without communicating with each other and therefore without knowing the secret, before or after the transition. Note that secure function computation (e.g., [10]) requires communication whenever inputs should be processed, while we require transition with no internal communication.

Our contributions. We define and present three reactive k-secret solutions. The first solution is based on the Chinese remainder, the second is based on polynomial representation, and the third uses replication of states. In the first solution we allow the addition arithmetic operation as a possible transition, where each share of the secret is modified according to the added value, without collecting the global secret value. The second solution supports both arithmetic addition and multiplication of the secret by a given input. The last solution implements a general I/O automaton, where the transition to the next step is performed according to the input event accepted by the swarm.

To avoid compromising the global secret of the swarm, the participants maintain only a share of the secret. In the Chinese remainder scheme, the participant share of the global value reveals partial information on the secret value. The polynomial based scheme assumes unbounded secret share values, which enable it to ensure that no information is revealed to the swarm members. Note that the shares of the polynomial based solution have number of bits that is approximately the number of bits of the secret, while the *total* number of bits of the shares of the Chinese remainder is approximately the number of bits required to describe the secret. The third solution replicates states of a given automaton and distributes several distinct replicas to each participant in the swarm. The relative majority of the distributed replicas represent the state of the swarm. The participant changes the states of all the replicas it maintains according to the global input arriving to the swarm. In this case, general automaton can be implemented by the swarm, revealing only partial knowledge on the secret of the swarm.

We remark that it is also possible to device Vandermonde matrix based scheme that supports other operations such as the bitwise-xor operation of the secret shares. In this case the secret is masked by a random number, and operations over shares are according to the relevant portions of the global input.

Paper organization. The system settings are described in Section 2. The k-secret addition implementation that is based on the Chinese remainder appears in Section 3. The solution that supports addition and multiplication by a number is the polynomial-based solution presented is Section 4. The I/O automaton implementation appears in Section 5. Finally, conclusions appear in Section 6. Some of the details are omitted from this extended abstract and can be found in [6].

2 Swarm Settings

A *swarm* consists of at least n processes (executed by, say, Unmanned Aerial Vehicles UAVs, mobile-sensors, processors) which receive inputs from the outside environment simultaneously[1]. The swarm as a unit holds a secret, where shares of the secret are distributed among the swarm members in a way that at least k are required to reveal the secret (some of our schemes require more than k), and any less than k shares can not reveal the secret. Yet, in some of our schemes the shares may imply some information regarding the secret, namely, the secret can be guessed with some positive probability < 1 greater than the probability of a uniform guess over the secret domain. We consider both listening adversary and Byzantine adversary, and present different schemes used by the processes to cope with them. We assume that at most $f < k$ of the n processes may be captured and compromised by an adversary. Communication among the processes in the swarm is avoided or performed in a safe land, alternatives of more expensive secure communication techniques are also mentioned.

Reactive k-secret counting. Assume that we have a swarm, which consists of n processes. The task of the swarm is to manage a global value called *counter*, which is updated according to the swarm input events. The value of the counter is the actual secret of the swarm. Each swarm member holds a share of the global counter in a way that any k or more members may reveal the secret with some positive probability Pr_k, yet any less than k members fail to reveal it.

Any event sensed by the processes is modeled as a system input. The swarm receives *inputs* and sends *outputs* to the outside environment. An input to the swarm arrives at all processes simultaneously. The output of the swarm is a function of the swarm state. There are two possible assumptions concerning the swarm output, the first, called *threshold accumulated output*, where the swarm outputs only when at least a predefined number of processes have this output locally. The second means to define the swarm output is based on secure internal communication within the swarm, the communication takes place when the local state of a process indicates that a swarm output is possible[2]. In the sequel we assume the *threshold accumulated output* where the adversary cannot observe outputs below the threshold. Whenever the output is above the threshold, the adversary may observe the swarm output together with the outside environment, and is "surprised" by the non anticipated output of the swarm.

We consider the following input actions, defined for our first solution:

• *set(x):* Sets the secret share with the value x. The value x is distributed in a secure way to processes in the swarm, each process receives a secret share x. This operation is either done in a safe land, or uses encryption techniques.

[1] Alternatively, the processes can communicate the inputs to each other by atomic broadcast or other weaker communication primitive.

[2] In this case, one should add "white noise" of constant output computations to mask the actual output computations.

• *step(δ):* Increments (or decrements in case δ is negative) the current counter value by δ. The processes of the swarm independently receive the input δ.
• *regain consistency:* Ensures that the processes carry the current counter value in a consistent manner. We view the execution of this command as an execution in a safe land, where the adversary is not present. The commands are used for reestablishing security level. Recovery and preparation for succeeding obstacles is achieved, by redistributing the counter shares, in order to prepare the swarm to cope with future joins, leaves, and state corruption.

The processes in the swarm communicate, if possible creating additional processes instead of the processes that left, and redistribute the counter value. This is the mechanism used to obtain a proactive security property.

We assume that the number of processes *leaving* the swarm between any two successive *regain consistency* actions, is bounded by lp. The operation taken by a leaving process is essentially an erase of its memory[3].

• *join request:* A process requests to join the swarm.
• *join reply:* A process reply to a join request of another process, by sending the joining process a secret share.

The adversary operations are:

• *listening:* Listening to all communication, but cannot send messages.
• *capture a process:* Remove a process from the swarm and reveal a snapshot of the memory of a process[4]. The adversary can invoke this operation at most $f = k - 1$ times between any two successive global resets of the swarm secret. Reset is implemented by using the set input actions.
• *corrupt process state:* The adversary is capable of changing the state of a process. In this case, the adversary is called a Byzantine adversary. Byzantine adversary also models the case in which transient faults occur, e.g., causing several of the processes not to get the same input sequence. We state for each of our schemes the number of times the adversary can invoke this operation between any two successive regain consistency operations.

3 Reactive *k*-Secret Counting – The Chinese Remainder Solution

According to the Chinese Remainder Theorem (CRT), any positive integer m is uniquely specified by its residue modulo relatively prime numbers $p_1 < ... < p_l$, where $\prod_{i=1}^{l-1} p_i \leq m < \prod_{i=1}^{l} p_i$ and $p_1 < p_2 < ... < p_l$. We use the CRT for defining the swarm's global counter, denoted by GC, which is the actual swarm secret.

[3] One may wish to design a swarm in which the members maintain the population of the swarm, in this case, as an optimization for a mechanism based on secure heart-bits, a leaving process may notify the other members on the fact it is leaving.
[4] In the sequel we assume that a joining process reveals information equivalent to a captured process, though, if it happen that the (listening) adversary is not presented during the join no information is revealed.

Using a Chinese remainder counter. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_l\}$, such that $p_1 < p_2 < ... < p_l$, is the set of l relatively prime numbers which defines the global counter GC. The integer values of GC run from 0 to GC_{max}, where $GC_{max} = \prod_{i=1}^{l} p_i - 1$.

A *counter component* is a pair $\langle r_i, p_i \rangle$, where $r_i = GC \bmod p_i$ and $p_i \in \mathcal{P}$. The swarm's global counter GC can be denoted by a sequence of l counter components $\langle \langle r_1, p_1 \rangle, \langle r_2, p_2 \rangle, .., \langle r_l, p_l \rangle \rangle$, the *CRT-representation* of GC, or $\langle r_1, r_2, .., r_l \rangle$, when $\mathcal{P}$ is known. Note that this representation implies that GC can hold up to $\prod_{i=1}^{l} p_i$ distinct values. A *counter share* is simply a set of distinct counter components.

We assume that there is a lower bound p_{min} on the relatively prime numbers in $\mathcal{P}$ such that $p_{min} < p_1 < p_2 < ... < p_l$.

For a given $\mathcal{P} = \{p_1, p_2, \ldots, p_l\}$ and $GC = \langle \langle r_1, p_1 \rangle, \langle r_2, p_2 \rangle, .., \langle r_l, p_l \rangle \rangle$, we distribute the counter GC among the n processes in a way that (a) k or more members may reveal the secret with some probability, yet (b) any fewer than k members fail to reveal it.

In order to support simple join input actions, we use the CRT-representation of GC in a way that each process holds a counter share of size $s = \lfloor \frac{l}{k} \rfloor$, namely, s counter components out of l. Let Pr_m denote the probability that all the l components of GC are present in a set of m distinct counter shares, each of size s as specified. We now compute Pr_m. For any $0 \leq m < k$ it holds that $Pr_m = 0$, since at least one counter component is missing. For $m \geq k$ it holds that $Pr_m = [1 - (1-p)^m]^l$, where p is the probability of a counter component to be chosen. As the components are chosen with equal probability out of the l components of GC, it holds that $p = \frac{s}{l} \approx \frac{1}{k}$. Assuming k divides l, it holds that $p = \frac{s}{l} = \frac{1}{k}$. The probability that a certain counter component appears in one of the m counter shares is $1 - (1-p)^m$. Hence, the probability that no component is missing is $[1-(1-p)^m]^l$. Therefore, $Pr_m = [1-(1-p)^m]^l$. Thus, the expected number m of required partial counters is a function of $n,\ l$, and k.

Note that when the GC value is incremented (decremented) by δ, each counter component $\langle r_i, p_i \rangle$ of GC is incremented (decremented) by δ modulo p_i. For example, let $\mathcal{P} = \{2, 3, 5, 7\}$ ($p_1 = 2,\ p_2 = 3,\ p_3 = 5,\ p_4 = 7$), $l = 4$ and $GC = 0$. The CRT-representation of GC is $\langle \langle 0, 2 \rangle, \langle 0, 3 \rangle, \langle 0, 5 \rangle, \langle 0, 7 \rangle \rangle$ or $\langle 0, 0, 0, 0 \rangle$ for the given set of primes $\mathcal{P}$. After incrementing the value of GC by one, it holds that $GC = \langle 1, 1, 1, 1 \rangle$. Incrementing by one again, results in $GC = \langle 0, 2, 2, 2 \rangle$, then $\langle 1, 0, 3, 3 \rangle$, $\langle 0, 1, 4, 4 \rangle$, $\langle 1, 2, 0, 5 \rangle$, and so on.

Next we describe the way the Chinese remainder counter supports the required input actions as appears in Figure 1.

Line-by-line code description. The code in Figure 1 describes input actions of process i. Each process i has a share of s counter components: s relatively prime numbers $primes_i[1..s]$ and s relative residues $residues_i[1..s]$, where $residues_i[j] = GC \bmod primes_i[j]\ \forall j = 1..s$.

Each input action includes a message of the form $\langle type,\ srcid,\ destid,\ parameters \rangle$, where *type* is the message type indicating the input action type, *srcid* is the identifier of the source process, *destid* the identifier of the destination

```
1  set_i(⟨set, srcid, i, share⟩)
2      for j = 1..s
3          primes_i[j] ⟵ getPrime(share, j)
4          residues_i[j] ⟵ getResidue(share, j)

5  step_i(⟨stp, srcid, i, δ⟩)
6      for j = 1..s do
7          residues_i[j] ⟵ (residues_i[j] + δ) mod primes_i[j]

8  regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
9      leaderId ⟵ leaderElection()
10     if leaderId = i then
11         globalCounterComponents_i ⟵ listenAll(⟨rgn_rply, i, j, share⟩)
12         if size(globalCounterComponents_i) < l then
13             globalCounterComponents_i ⟵ initGlobalCounterComponents()
14         for every process id j in the swarm do:
15             share ⟵ randomShare(globalCounterComponents)
16             send(⟨set, i, j, share⟩)
17         globalCounterComponents_i ⟵ ∅
18     else
19         send(⟨rgn_rply, i, leaderId, ⟨primes_i[1..s], residues_i[1..s]⟩⟩)

20 regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
21     if leaderId = i then
22         globalCounterComponents_i ⟵ globalCounterComponents_i ∪ {share}

23 joinRequest_i(⟨join_rqst, srcid, i⟩)
24     sentPrimes ⟵ ∅
25     while |sentPrimes| < s do
26         waitingTime ⟵ random([1..maxWaiting(n)])
27         while waitingTime not elapsed do
28             listen(⟨join_rply, i, pid, p, r⟩)
29             sentPrimes = sentPrimes ∪ {p}
30         if |sentPrimes| < s then
31             p′ ⟵ getRandom(primes_i \ sentPrimes)
32             r′ ⟵ getAssociatedResidue(p′)
33             send(⟨join_rply, i, srcid, p′, r′⟩)

34 joinReply_i(⟨join_rply, srcid, i, p, r⟩)
35     shareSize ⟵ size(primes_i)
36     if shareSize < s then
37         if p ∉ primes_i then
38             primes_i[shareSize] ⟵ p
39             primes_i[shareSize] ⟵ r
```

Fig. 1. Chinese Remainder, program for swarm member i

process and further parameters required for the actions executed as a result of the input action.

• *set:* On *set*, process i receives a message of type *set*, indicating the *set* input action and a counter share *share*, namely a set of s counter components (line 1). Process i sets $primes_i$ and $residues_i$ with the received primes and relative residues of the received counter share *share* (lines 2–4).
• *step:* On *step*, process i receives a message of type *stp*, indicating the *step* input action, and an increment value δ, which may be negative (line 5). See a similar technique in [7]. The δ value indicates a positive or negative change in the global counter that affects all the counter shares.

Incrementing (or decrementing) the global counter by δ is done by incrementing (or decrementing) each residue r_{i_j} in the counter share of process i by δ

modulo p_{i_j} such that $residues_i[j]$ is set with $(residues_i[j] + \delta)$ mod $primes_i[j]$ (lines 6,7).

• *regainConsistencyRequest:* On *regainConsistancyRequest*, the processes are assumed to be in a safe place without the threat of any adversary (alternatively, a global secure function computation technique is used).

Process i receives a message of type *rgn_rqst* (line 8) which triggers a leader election procedure (line 9). Once the leader is elected, it is responsible for distributing the global counter components to the swarm members. If process i is the leader (line 10) it first listens to regain consistency reply messages, initializing the set *globalCounterComponents* with the counter components received from other swarm members (line 11).

If the number of distinct global counter components is smaller than s, i.e., some of the global counter components are missing, then process i initializes *globalCounterComponents* with the set of components calculated by the method *initGlobalCounterComponents*() (lines 12,13). This method sets the values of the global counter GC by setting l distinct primes and l relative residues.

Having set the global counter components, process i (the leader) randomly chooses a share of size s (out of l) components and sends it to a swarm member. Note that there is also a straightforward deterministic way to distribute the shares, or alternatively check the result of the random choice. The random share is chosen with equal probability for every swarm member and sent in a message of type *set* (lines 14-16).

After the shares are sent, the set *globalCounterComponents* is initialized as an empty set, to avoid revealing the counter in case the leader is later compromised (line 17). In case process i is not the leader, it sends its share to the leader (lines 18,19).

• *regainConsistencyReply:* On *regainConsistancyReply*, the processes are assumed to be in a safe place without the threat of any adversary.

Process i receives a message of type *rgn_rply* with the counter share of a process whose identifier is *srcid* (line 20). If process i is the leader, then it adds the components of the received share to its own set of *globalCounterComponents* (lines 21,22).

• *joinRequest:* An input message of type *join_rqst* indicates a request by a new process with identifier *srcid* to join the swarm (line 23). Process i holds a set *sentPrimes* of primes which were sent by other processes in a join reply message. This set is initially empty (line 24).

The join procedure is designed to restrict the shares the (listening) adversary may reveal during the join procedure to the shares assigned to the joining process. Thus, if the number of distinct primes which were sent to the joining process is at least s, then process i should not reply to the join request (line 25).

Otherwise, it sets *waitingTime* with a random period of time, which is a number of time units within the range 1 and $maxWaiting(n)$; where $maxWaiting$ is a function which depends on the number of swarm members n and the time unit size (line 26).

During the random period of time $waitingTime$, process i listens to join replies sent by other processes. Each reply includes a prime p and its relative residue r. While listening, process i adds the sent prime p to the set of primes $sentPrimes$ (lines 27–29). After the $waitingTime$ elapsed, process i checks if at least s distinct counter components were sent back to the joining process (line 30). If not, it randomly chooses a prime number p' out of the primes that appears in its share but not in $sentPrimes$. It then sends back to process $srcid$ a join reply with the random counter component $\langle p', r' \rangle$, where r' is the residue associated with p', namely $r' = GC \bmod p'$ (lines 31–33).

We assume that at most one sender may succeed in sending the reply. If one has failed, process i knows which counter component was successfully sent. Note that the counter components can be encrypted. In that case, the $join\_rqst$ message includes a public key. Otherwise, we regard each join as a process capture by the adversary.

• *joinReply:* Process i receives an input message of type $join\_rply$, which indicates a reply for a join request by a process joining the swarm. The message includes a counter component: Prime p and its relative residue r (line 34).

Process i sets the $shareSize$ with the size of $primes_i$ and indicates the current size of its current share, which should eventually be s (line 35). If $shareSize$ is smaller than s (line 36), then process i should not ignore incoming join replies since it is missing counter components. Process i checks whether the received prime p was already received. If so, process i adds the received counter components by adding p to $primes_i$ and r to $residues_i$ (lines 37–39).

Theorem 1. *In any execution in which the adversary captures at most $k-1$ processes, the probability of the adversary guessing the secret, i.e., guessing the value of the global counter, is bounded by $\frac{1}{p_{min}}$.*

Byzantine adversary and error correcting. We now turn to considering the case of the Byzantine adversary, in which some errors take place, such as input not received by all swarm members. Let m be any positive integer, where $\prod_{i=1}^{l-1} p_i \leq m < \prod_{i=1}^{l} p_i$ and $p_1 < p_2 < ... < p_l$. By the CRT, m is uniquely specified by its residues modulo relatively prime numbers $p_1 < ... < p_l$.

The integer m can be represented by $l + l_0$ ($l_0 > 0$) residues modulo relatively prime numbers $p_1 < ... < p_{l+l_0}$. Clearly, this representation is not unique and uses l_0 redundant primes. The integer m can be considered a code word, while the extended representation (using $l+l_0$ primes) yields a natural *error-correcting code* [8].

The error correction is based on the property that for any two integers $m, m' < \prod_{i=1}^{l+l_0} p_i$ the sequences $\{(m \bmod p_1), ..., (m \bmod p_{l+l_0})\}$ and $\{(m' \bmod p_1), ..., (m' \bmod p_{l+l_0})\}$ differ in at least l_0 coordinates.

On the presence of errors, the primes may also be faulty. For that, let us assume that each process keeps the whole set $\mathcal{P}$ instead of only a share of it. Let us also assume that $\mathcal{P}$ is of size $l + l_0$, where l_0 primes are redundant. Under this assumption, we can update the *regainConsistency* action, so that the processes first agree on $\mathcal{P}$ by a simple majority function and only then agree on the residues

$\langle r_1, r_2, ..., r_{l+l_0} \rangle$ matching the relatively primes $\{p_1 < p_2 < ... < p_{l+l_0}\} = \mathcal{P}$. In that case, the number of Byzantine values or faults, modeled by f, is required to be less than the majority and less than $\frac{l_0}{2}$.

Once $\mathcal{P}$ is agreed on, the received counter components $\langle p_j, r_j \rangle$ where $p_j \notin \mathcal{P}$ are discarded, while the rest of the components are considered candidates to be the real global counter components. The swarm then needs to agree on the residues and again, it is done by a simple majority function executed for every residue out of the $l + l_0$ residues. After the swarm has agreed on the primes and the residues (and in fact, on the counter), the consistency of the counter components can be checked using Mandelbaum's technique [12]. If $f < l_0$ errors have occurred, then they can be detected. Also, the number of errors which can be corrected is $\lfloor l_0/2 \rfloor$.

Chinese remainder solution with single component share. In this case, we have l relatively prime numbers in $\mathcal{P}$ and each secret share is a single pair $\langle r_i, p_i \rangle$, where $p_i \in \mathcal{P}$ and $r_i = GC \bmod p_i$. Using Mandelbaum's technique [12] we may distribute $l \geq k$ shares that represent a value of a counter defined by any k of them. So in case of the listening adversary with no joins, any k will reveal the secret and less than k will not. This distribution also supports the case of Byzantine/Transient faults, where more than k should be read in order to reveal the correct value of the secret. In such a case, a join can be regarded as a transient fault, having the joining process choose its share to be a random value.

4 Reactive *k*-Secret – Counting/Multiplying Polynomial-Based Solution

Here we consider a global counter, where its value can be multiplied by some factor, as well as increased (decreased) as described in Section 3. The global counter is based on Shamir's (k, n)-threshold scheme [13], according to which, a secret is divided into n shares in a way that any k or more shares can reveal it yet no fewer than k shares can imply any information regarding the secret. Given $k + 1$ points in the 2-dimensional plane $(x_0, y_0), \ldots (x_k, y_k)$, where the values x_i are distinct, there is one and only one polynomial $p(x)$ of degree k such that $p(x_i) = y_i$ for $i = 0..k$. The secret, assumed to be a number S, is encoded by $p(x)$ such that $p(0) = S$. In order to divide the secret S into n shares $S_1, \ldots, S_n$, we first need to construct the polynomial $p(x)$ by picking k random coefficients $a_1, \ldots, a_k$ such that $p(x) = S + a_1x^1 + a_2x^2 + \ldots + a_kx^k$. The n shares $S_1, \ldots, S_n$ are pairs of the form $S_i = \langle i, p(i) \rangle$. Given any subset of k shares, $p(x)$ can be found by interpolation and the value of $S = p(0)$ can be calculated.

Our polynomial-based solution encodes the value of the global counter GC, which is the actual secret shared by the swarm members. The global counter GC is represented by a polynomial $p(x) = GC + a_1x^1 + a_2x^2 + \ldots + a_lx^l$, where $a_1, \ldots, a_l$ are random. Now, the l counter components are the points $\langle i, p(i) \rangle$ for $i = 1..l$. Instead of each secret share being a single point, a share is a tuple of $s = \lfloor \frac{l}{k} \rfloor$ such points. This way, compromising at most $k-1$ processes ensures that at least one point is missing and therefore the polynomial $p(x)$ cannot be calculated.

Having each process holding a single point as in Shamir's scheme, implies a complicated join action since it requires collecting all the swarm members' shares and computing the polynomial $p(x)$ in order to calculate a new point for the new joining process. Such action should be avoided under the threat of an adversary. A tuple of $s = \lfloor \frac{l}{k} \rfloor$ points implies a safe join action. The processes share their own points with the new joining process and there is no need to collect all the points.

Lemma 1. *Let $P(x)$ be a polynomial of degree d. Given a set of $d+1$ points $(x_0, y_0), \ldots (x_d, y_d)$, where $P(x_i) = y_i$ for $i = 0, .., d$ and a number δ. The polynomial $Q(x)$, also of degree d, where $Q(x_i) = y_i + \delta$, equals to $P(x) + \delta$.*

Lemma 2. *Let $P(x)$ be a polynomial of degree d. Given a set of $d+1$ points $(x_0, y_0), \ldots (x_d, y_d)$, where $P(x_i) = y_i$ for $i = 0, .., d$ and a number μ. The polynomial $Q(x)$ of degree d, where $Q(x_i) = y_i \cdot \mu$, equals to $P(x) \cdot \mu$.*

According to lemma 1 adding δ to $y_0, \ldots y_l$, where $P(i) = y_i$ for $i = 0, .., l$, results in a new polynomial $Q(x)$ where $Q(x) = P(x) + \delta$. Hence, increasing (decreasing) the second coordinate of the counter shares by δ increases (decreases) the secret by δ as well, since $Q(0) = P(0) + \delta$. Similarly, according to lemma 2 multiplication of the second coordinate in some factor μ implies the multiplication of the secret value in μ.

The code for the polynomial based solution is omitted from this extended abstract and cane be found in [6]. The procedure for each input actions are very similar to the input actions in the Chinese remainder counter, see Figure 1. In this case, the secret shares are tuples of s points given in two arrays $xCoords[1..s]$ and $yCoords[1..s]$, matching the x and y coordinates, respectively. These arrays replace the $primes[1..s]$ and $residues[1..s]$ arrays in the Chinese remainder counter. Another difference is that the step operation has an additional parameter *type*, which defines whether to increment (decrement) or multiply the counter components by δ.

Theorem 2. *In any execution in which the adversary captures at most $k-1$ processes, the adversary does not reveal any information concerning the secret.*

Byzantine adversary and error correcting. Analogously to the Chinese remainder case, we can design a scheme that is robust to faults. Having n distinct points of the polynomial $p(x)$ of degree l, the Berlekamp-Welch decoder [15] can decode the secret as long as the number of errors e is less than $(n-l)/2$.

Polynomial-based solution with single component share. Using Berlekamp-Welch technique [15] we may distribute $n \geq k$ shares that represent a value of a counter defined by any k of them. So in the case of the listening adversary with no joins, any k will reveal the secret, and fewer than k will not. Similarly to the Chinese remainder solution, this secret share distribution also supports the case of Byzantine/Transient faults, where more than k should be read to reveal the correct value of the secret. Again, a join can be regarded as a transient fault, having the joining process choose its share to be a random value.

5 Virtual Automaton

We would like the swarm members to implement a virtual automaton where the state is not known. Thus, if at most f, where $f < n$, swarm members are compromised, the global state is not known and the swarm task is not revealed.

In this section we present the scheme assuming possible errors, as the error free is a straightforward special case.

We assume that our automaton is modeled as an I/O automaton [11] and described as a five-tuple:

• An action signature $sig(A)$, formally a partition of the set $acts(A)$ of actions into three disjoint sets $in(acts(A))$, $out(acts(A))$ and $int(acts(A))$ of input actions, output actions, and internal actions, respectively. The set of local controlled actions is denoted by $local(A) = out(A) \cup int(A)$.
• A set $states(A)$ of states.
• A nonempty set $start(A) \subseteq states(A)$ of initial states.
• A transition relation $steps(A) : states(A) \times acts(A) \longrightarrow states(A)$, where for every state $s \in states(A)$ and an input action π there is a transition $(s, \pi, s') \in steps(A)$.
• An equivalence relation $part(A)$ partitioning the set $local(A)$ into at most a countable number of equivalence classes.

We assume that the swarm implements a given I/O automaton A. The swarm's *global state* is the current state in the execution of A. Each process i in the swarm holds a tuple $cur\_state_i = \langle s_{i_1}, s_{i_2}, \ldots, s_{i_m} \rangle$ of m distinct states, where $s_{i_j} \in states(A)$ for all $j = 1..m$ and at most one of the m states is the swarm's global state. Formally, the swarm's global state is defined as the state which appears in at least threshold T out of n *cur_state* tuples ($T \leq n$). If there are more than one such states, then the swarm's global state is a predefined default state.

The *output* of process i is a tuple $out_i = \langle o_{i_1}, o_{i_2}, \ldots, o_{i_m} \rangle$ of m output actions, where $o_{i_j} \in out(acts(A))$ for all $j = 1..m$. The swarm's *global output* is defined as the result of the output action which appears in at least threshold T out of n members' output.

We assume the existence of a devices (sensors, for example) which receives the output of swarm members (maybe in the form of directed laser beams) and thus can be exposed to identify the swarm's global output by a threshold of the members outputs.

We assume an adversary which can compromise at most $f < n$ processes between two successive global reset operations of the swarm's global state. We assume that the adversary knows the automaton A and the threshold T. Therefore, when compromising f processes it can sample the *cur_state* tuples of the compromised processes and assume that the most common state, i.e., appears as many times in the compromised *cur_state* tuples, is most likely to be the global state of the swarm.

Consider the case in which $f = 1$ and $T = \lfloor n/2 + 1 \rfloor$. If $|cur\_state| = 1$ (i.e., there is a single state in *cur_state*), then an adversary which compromises process i, knows the state $s_{i_1} \in cur\_state_i$. The probability that s_{i_1} is the swarm's global

state is at least $\frac{T}{n}$ and since T is a lower bound, the probability may reach 1 when all shares are identical. If $|cur\_state| = 2$, then an adversary which compromises process i, knows the state $s_{i_1}, s_{i_2} \in cur\_state_i$. The probability that either of the states s_{i_1} or s_{i_2} is the swarm's global state is at least $\frac{T}{n}$. Since there is no information on which state of the two is the most likely to be the swarm's global state, the only option for the adversary is to arbitrarily choose one of the two states with equal probability. Therefore, the probability of revealing the swarm's global state is at least $\frac{T}{2n}$ and at most $\frac{T}{n}$ in case $|cur\_state| = 2$. Generally, if $|cur\_state| = m$, then the probability of revealing the swarm's global state is at least $\frac{T}{m \cdot n}$, and at most $\frac{T}{(m-1) \cdot n}$ for $f = 1$. As the number of states in *cur_state* increases, the probability to reveal the swarm's global state decreases.

We consider the following input actions:

• *set*($\langle s_{i_1}, \ldots, s_{i_m} \rangle$)*:* Sets *cur_states* with the given tuple. The tuples are distributed in a way that at least $T + f + lp$ of them contain the swarm's global state. Thus, even if f shares are corrupted and lp are missing because of the leaving processes, the swarm threshold is respected. Moreover, in order to ensure uniqness of the global state in the presence of corruptions and joins, any other state has less than $T - f$ replicas.
• *step*(δ)*:* Emulates a step of the automaton for each of the states in cur_state_i. By the end of the emulation each process has output. Here, δ is any possible input of the simulated automaton.
• *regain consistency:* Ensures that there are at least $T + f + lp$ members, whose *cur_states* tuples include the current state of A. Any other state has less than $T - f$ replicas.
• *join:* A process joins the swarm, and constructs its *cur_states* tuple by randomly collecting states from other processes. Note that the scheme benefits from smooth joins, since the number f that includes the join operations is taken in consideration while calculating the swarm's global state upon regain consistency operation. That is, a threshold of T is required for a state in order to be the swarm's global state. Therefore, in case swarm members maintain the population of the swarm (updated by joins, leaves and possibly by periodic heartbits) a join may be simply done by sending a join request message, specifying the identifier of the joining process. However, the consistency of the swarm will definitely benefit if shares are uniformly chosen for the newcomers. In this way, if the adversary was not listening during the join procedure, there is high probability that the joining processes will assist in encoding the current secret.

Line-by-line code description. The code in Figure 2 describes input actions of process i. Each process i has an m-tuple cur_state_i of m states in $states(A)$, where at most one of them is the swarm's global state.

• *set:* On input action *set*, process i receives a message of type *set* and an m-tuple of distinct states in $states(A)$ (line 1). It then sets its tuple cur_state_i with the received tuple (line 2).
• *step:* On input action *step*, process i receives a message of type *stp* and δ, which is an input parameter for the I/O automaton (line 3). For every state s_{i_j} in

```
1  set_i(⟨set, srcid, i, ⟨s_i1, ..., s_im⟩⟩)
2      cur_state_i ⟵ ⟨s_i1, ..., s_im⟩

3  step_i(⟨stp, srcid, i, δ⟩)
4      for j = 1..m do
5          ⟨s'_ij, o_ij⟩ ⟵ follow the transaction in steps(A) for s_ij and δ
6      next_state ⟵ ⟨s'_i1, ..., s'_im⟩
7      output_acts ⟵ ⟨o'_i1, ..., o'_im⟩
8      executeOutputActions(output_acts)
9      cur_state_i ⟵ next_state

10 regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
11     leaderId ⟵ leaderElection()
12     if leaderId = i then
13         allStateTuples_i ⟵ collectAllStates()
14         candidates ⟵ mostPopularStates(allStateTuples)
15         if |candidates| == 1 then
16             globalState ⟵ first(candidates)
17         else
18             globalState ⟵ defaultGlobalState
19         distributeStateTuples(globalState)
20         allStateTuples_i ⟵ ∅
21         delete candidates
22     else
23         send(⟨rgn_rply, i, leaderId, cur_state_i, ⟩)

24 regainConsistencyReply_i(⟨rgn_rply, srcid, i, stateTuple⟩)
25     if leaderId = i then
26         allStateTuples_i ⟵ allStateTuples_i ∪ {stateTuple}

27 joinRequest_i(⟨join_rqst, srcid, i⟩)
28     addMember(srcid)
```

Fig. 2. Virtual automaton, program for swarm member i

cur_state_i process i emulates the automaton A by executing a single transaction on s_{i_j} and δ (line 5). As a result, there is a new state s'_{i_j} and an output action o_{i_j}. Process i initializes a tuple $next\_state$ of all the new states s'_{i_j} for all $j = 1..m$ (line 6) and the resulting output actions o'_{i_j} for all $j = 1..m$ (line 7). It then executes the output actions in $output\_acts$ (line 8) and finally, it updates cur_state_i to be the tuple of new states $next\_state$ (line 9).

• *regainConsistencyRequest:* On input action *regainConsistency* the processes are assumed to be in a safe land with no threat of any adversary. Process i receives a message of type rgn_rqst from process identified by $srcid$ (line 10).

The method $leaderElection()$ returns the process identifier of the elected leader (line 11). If process i is the leader, then it should distribute state tuples using *set* input actions in a way that at least $T + f$ swarm members have tuples that include the global state and all other states appear no more than T times. Possibly by randomly choosing shares to members, such that the probability for assigning the global state share to a process is equal to, or slightly greater than, $T/n + f/n$ while the probability of any other state to be assigned to a process is the same (smaller) probability.

First, the leader collects all the state tuples (line 13) and then executes the method $mostPopularStates()$ in order to find the candidates to be the swarm's global state (lines 14). If there is a single candidate (line 15), then it is the global

state and *globalState* is set with the first (and only) state in *candidates* (line 16). In case there is more than one candidate (line 17), the leader sets *globalState* with a predefined default global state (line 18).

The leader then distributes the state tuples (line 19) and deletes both the collected tuples *allStateTuples* and the candidates for the global state *candidates* (lines 20,21). If process i is not the leader, then it sends its cur_state_i tuple to the leader (lines 22,23).

• *regainConsistencyReply:* On input action *regainConsistencyReply* the processes are also assumed to be in a safe land. Process i receives a message of type *rgn_rply*, which is a part of the regain consistency procedure. The message includes the identifier *srcid* of the sender and the sender's state tuple (line 24). If process i is the leader (line 25), then it adds the received tuple to the set $allStateTuples_i$ of already received tuples. Otherwise, it ignores the message.
• *joinRequest:* On input action, *joinRequest* process i receives a message of type *join_rqst* from a process identified by *srcid*, which is asking to join the swarm (line 27). Process i executes the method $addMember(srcid)$, which adds *srcid*, the identifier of the joining process, to its population list of processes in the swarm.

6 Conclusions

We have presented three (in fact four, including the Vandermonde matrix based scheme) approaches for reactive k-secret sharing that require no internal communication to perform a transition.

The two first solutions maybe combined as part of the reactive automaton to define share of the state, for example to enable an output of the automaton whenever a share value of the counter is prime. Thus the operator of the swarm may control the output of each process by manipulating the counter value, e.g., making sure the counter secret shares are never prime, until a sufficient number and combination of events occurs.

We believe that such a distributed manipulation of information without communicating the secret shares, that is secure even from the secret holders, should be further investigated. At last, the similarity in usage of Mandelbaum and Berlekamp-Welch techniques may call for arithmetic generalization of the concepts.

References

1. Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., Nolte, T.: Virtual Stationary Automata for Mobile Networks. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, Springer, Heidelberg (2006) Also invited paper in Forty-Third Annual Allerton Conference on Communication, Control, and Computing. Also, Brief announcement. In: PODC 2005. Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing, p. 323 (2005) Technical Report MIT-LCS-TR-979, Massachusetts Institute of Technology (2005)

2. Dolev, S., Gilbert, S., Lynch, A.N., Schiller, E., Shvartsman, A., Welch, J.: Virtual Mobile Nodes for Mobile Ad Hoc Networks. In: DISC 2004. International Conference on Principles of DIStributed Computing, pp. 230–244 (2004) Also Brief announcement. In: PODC 2004. Proc. of the 23th Annual ACM Symp. on Principles of Distributed Computing (2004)
3. Dolev, S., Gilbert, S., Lynch, N.A., Shvartsman, A., Welch, J.: GeoQuorum: Implementing Atomic Memory in Ad Hoc Networks. Distributed Computing 18(2), 125–155 (2003)
4. Dolev, S., Gilbert, S., Schiller, E., Shvartsman, A., Welch, J.: Autonomous Virtual Mobile Nodes. In: DIALM/POMC 2005. Third ACM/SIGMOBILE Workshop on Foundations of Mobile Computing, pp. 62–69 (2005) Brief announcement. In: SPAA 2005. Proc. of the 17th International Conference on Parallelism in Algorithms and Architectures, p. 215 (2005) Technical Report MIT-LCS-TR-992, Massachusetts Institute of Technology (2005)
5. Dolev, S., Lahiani, L., Lynch, N., Nolte, T.: Self-Stabilizing Mobile Location Management and Message Routing. In: Tixeuil, S., Herman, T. (eds.) SSS 2005. LNCS, vol. 3764, pp. 96–112. Springer, Heidelberg (2005)
6. Dolev, S., Lahiani, L., Yung, M.: Technical Report TR-#2007-12, Department of Computer Science, Ben-Gurion University of the Negev (2007)
7. Dolev, S., Welch, L.J.: Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. Journal of the ACM 51(5), 780–799 (2004)
8. Goldrich, O., Ron, D., Sudan, M.: Chinese Remaindering with Errors. In: Proc. of 31st STOC. ACM (1999)
9. Kivelevich, E., Gurfil, P.: UAV Flock Taxonomy and Mission Execution Performance. In: Proc. of the 45th Israeli Conference on Aerospace Sciences (2005)
10. Kilian, J., Kushilevitz, E., Micali, S., Ostrovsky, R.: Reducibility and Completeness In Multi-Party Private Computations. In: FOCS 1994. Proceedings of Thirty-fifth Annual IEEE Symposium on the Foundations of Computer Science, Journal version in SIAM J. Comput. 29(4), 1189-1208 (2000)
11. Lynch, N., Tuttle, M.: An introduction to Input/Output automata, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 2(3), 219–246 (September 1989) Also Tech. Memo MIT/LCS/TM-373
12. Mandelbaum, D.: On a Class of Arithmetic and a Decoding Algorithm. IEEE Transactions on Information Theory 21(1), 85–88 (1976)
13. Shamir, A.: How to Share a Secret. CACM 22(11), 612–613 (1979)
14. Weiser, M.: The Computer for the 21th Century. Scientific American (September 1991)
15. Welch, L., Berlekamp, E.R.: Error Correcting for Algebraic Block Codes, U.S. Patent 4633470 (September 1983)