Ad Hoc Networks 10 (2012) 1291–1305

Contents lists available at SciVerse ScienceDirect

Ad Hoc Networks

journal homepage: www.elsevier.com/locate/adhoc

Secret swarm unit ☆
Reactive k-secret sharing

Shlomi Dolev [a,*], Limor Lahiani [a], Moti Yung [b]

[a] Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel
[b] Department of Computer Science, Columbia University, New York, NY, USA

ARTICLE INFO

Article history:
Received 21 December 2010
Accepted 29 March 2012
Available online 20 April 2012

Keywords:
Secret sharing
Mobile computing
Secure multi-party computation

ABSTRACT

Secret sharing is a 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 shares 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 reconstruct the secret and no $k-1$ or fewer can do so.

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.

1. Introduction

Secret sharing is a basic and fundamental technique [14]. Motivated by the high level of interest in the virtual automata abstraction and swarm computing, e.g., [4,3,2,5,7], we investigate an extension of the k-secret sharing scheme, in which the secret shares are modified on the fly, while maintaining the requirement that k or more shares may reconstruct the secret and no $k-1$ or fewer can reconstruct it.

There is great interest in pervasive ad hoc and swarm computing [15], particularly in swarming unmanned aerial vehicles (UAV) [10,5]. 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 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 swarm members are compromised and their secret shares are revealed. We would like the swarm members 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., [11]) requires communication whenever inputs should be processed, while we require transition with no internal communication.

1.1. Our contributions

We define and present four reactive k-secret schemes. The first three schemes are for the case in which the global secret of the swarm is some numeric number that can be modified according to inputs. The fourth scheme is for

☆ Partially supported by the Israeli Ministry of Science, Israel Science Foundation (Grant Number 428/11), Lynne and William Frankel Center for Computer Sciences and the Rita Altura Trust Chair in Computer Sciences. An extended abstract appeared in Indocrypt 2007 [8].

* Corresponding author.
E-mail addresses: dolev@cs.bgu.ac.il (S. Dolev), lahiani@cs.bgu.ac.il (L. Lahiani), moti@cs.columbia.edu (M. Yung).

1570-8705/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.adhoc.2012.03.011

the case of an I/O automaton, implemented by the swarm, where the current state of the automaton is the actual secret of the swarm. To avoid compromising the global secret of the swarm, the members maintain only a share of the secret.

The first polynomial scheme is based on Shamir's (k,n)-threshold scheme, the second is based on the Chinese Remainder Theorem (CRT) and the third is based on the Vandermonde matrix. The fourth and last solution uses replication of states for implementing a virtual I/O automaton with unknown state.

In all the solutions we suggest a way to modify the global secret by modifying only the secret shares without the need to collect them to reconstruct the global secret. In the polynomial-based and the CRT-based schemes we support arithmetic addition and multiplication operations as a possible modification of the secret. Still, these two implementations differ. In terms of total space used to encode the secret, the size of each share in the polynomial-based scheme is of the same order of the secret size, while the *total* size of the shares needed to define the secret in the Chinese remainder scheme is of the secret order. Also, in the Chinese remainder-based scheme, the secret share may reveal partial information on the global secret.

In the Vandermonde-based scheme, a predefined Vandermonde matrix is used to define the secret shares. This scheme is the only one that supports bitwise-*xor* operations among the non-automaton schemes.

The last scheme implements a general I/O automaton, where the transition to the next state is performed according to the input event received by the swarm. This scheme replicates states of a given automaton and distributes several distinct replicas to each swarm member. The relative majority of the distributed replicas represent the state of the swarm. A swarm member changes the states of all the replicas it maintains according to the global input received by the swarm. In this case, a general automaton can be implemented by the swarm, revealing only partial knowledge on the secret of the swarm.

1.2. Paper organization

The system settings are described in Section 2. The polynomial-based solution, which supports arithmetic addition and multiplication is presented in Section 3. The Chinese remainder-based solution appears in Section 4. The Vandermonde matrix-based solution appears in Section 5. Section 6 describes the I/O automaton implementation. Finally, conclusions appear in Section 7.

2. Swarm Settings

A *swarm* is a collection of processes (executed by, say, UnmannedAerial Vehicles UAVs, mobile-sensors, processors) that receive inputs from the outside environment simultaneously.[1] The swarm as a unit holds a secret, where shares of the secret are distributed among swarm members in a way that at least k are required to reconstruct the secret, and any fewer than k shares cannot reconstruct it. Yet, in some of our schemes the shares may imply additional information regarding the secret. Obviously, given a secret domain, the secret can be guessed with a uniform probability over the secret domain. We consider both *passive adversary* and *active (Byzantine) adversary*, and present different schemes used by the processes to cope with them. We assume that at most f of the n processes may be compromised or corrupted by an adversary, where $f < k$. Communication between the swarm members is avoided or performed in a safe land, alternatives of more expensive secure communication techniques may be used when communication is needed [11].

2.1. Reactive k-secret sharing – problem definition

Assume that we have a swarm, which initially consists of n processes. The task of the swarm is to manage a *global secret* and modify it without the need to reconstruct it first. Each swarm member holds a share of the global secret in a way that any fewer than k members fail to reconstruct the secret and at least k members may reconstruct it with some positive probability. In addition, all members can reconstruct the secret with probability 1.

2.2. Reactive k-secret sharing – general solution scheme

Any event sensed by the processes is modeled by 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 and the system inputs. 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 of defining 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 (similar to the secret maturity approach presented in [6]).

We consider the following input actions, to be implemented by each of our solutions:

- *set* (x): Sets the secret share with the value x. The value x is distributed in a secure way among processes of 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.

```
set_i(⟨set, srcid, i, share⟩)
    x_i ⟵ getX(share, 1)
    y_i ⟵ getY(share, 1)

step_i(⟨stp, srcid, i, op, δ⟩)
    if op == ADD
        y_i ⟵ y_i + δ
    else
        y_i ⟵ y_i * δ

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
        allSecretComponents_i ⟵ listenAll(⟨rgn_rply, j, i, share⟩)
        if size(allSecretComponents_i) < k then
            allSecretComponents_i ⟵ setDefaultSecret()
        for every process id j in the swarm do:
            new_share ⟵ getRandomShare(allSecretComponents, 1)
            send(⟨set, i, j, new_share⟩)
        ⟨(x_i, y_i)⟩ ⟵ getRandomShare(allSecretComponents, 1)
        allSecretComponents_i ⟵ ∅
    else
        send(⟨rgn_rply, i, leaderId, ⟨(x_i, y_i)⟩⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
        allSecretComponents_i ⟵ allSecretComponents_i ∪ share

joinRequest_i(⟨join_rqst, srcid, i⟩)
    replyWasSent ⟵ false
    waitingTime ⟵ random([1..maxWaiting(n)])
    while waitingTime not elapsed do
        replyWasSent ⟵ listen(⟨join_rply, j, srcid, component⟩)
    if replyWasSent = false then
        send(⟨join_rply, i, srcid, (x_i, y_i)⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    x_i ⟵ getX(component)
    y_i ⟵ getY(compoent)
```

Fig. 1. Polynomial-based solution with single component share. Program for swarm member *i*.

- *step*(δ): Modify the secret share, which results in modifying the global secret. The processes of the swarm independently receive the input δ, which is the modification parameter.
- *regain consistency request*: Request to redistribute the secret in order to ensure that the processes carry the current secret value in a consistent manner and to recover the secret if necessary. Also, the operation is required in order to cope with future joins, leaves, and state corruption. We assume that the execution followed by this input action is done in a safe land, where there is no threat of any adversary. The regain consistency mechanism is 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 n_{lp}. This number also includes failed processes since a failed process is considered a leaving process. The operation taken by a leaving process is essentially an erase of data related to the Swarm.[3]
- *regain consistency reply*: Reply to regain consistency request, which includes the updated secret share.
- *join request*: A process requests to join the swarm and receives join reply messages from other swarm members, to compose its own secret share.
- *join reply*: A process replies to a join request of another process, by sending the joining process a secret share.[4]

  We consider two types of adversaries:

[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-beats, a leaving process may notify the other members of the fact that it is leaving.

[4] Note that during a join operation, the communication is not intra-swarm communication since the swarm members communicate with a new joining process, which is yet to be a member of the swarm.

- *passive adversary*: can compromise at most $f < k$ processes and reveal their state.[5]
- *Active (Byzantine) adversary*: can reveal and corrupt the state of at most $f < k$ processes.
  Compromising or corruption can be invoked at most $f < k$ times between any two successive global resets of the swarm secret. A *global reset* of the secret can be implemented by using the *set* input actions to reset the secret shares or by executing a regain consistency operation, which redistributes the secret and may reset it to some predefined default secret.

3. Reactive *k*-secret sharing – polynomial-based scheme

Consider a swarm of initially n members and a global secret, denoted by *gs*, which is the actual secret of the swarm. The value of *gs* can be increased (decreased) by some integer value or multiplied by some integer factor. In our polynomial scheme, we use Shamir's (k,n)-threshold scheme [14] to encode the value of *gs*. That is done by using a polynomial $p(x)$ of degree $k-1$ over a finite field, such that $p(x) = a_0 + a_1x^1 + a_2x^2 + \cdots + a_{k-1}x^{k-1}$, where $a_1, \ldots, a_{k-1}$ are random and $a_0 = gs$.

A *secret component* is a pair (x,y), where $y = p(x)$ and $x \neq 0$. A set of n distinct secret components $(x_1,y_1), \ldots, (x_n,y_n)$ encodes the swarm's global secret *gs* in a way that any fewer than k secret components cannot reconstruct the polynomial $p(x)$ and any k secret components or more can do so and, hence, calculate the swarm's global secret *gs*. A *secret share* is simply a set (tuple) of distinct secret components.

Observation 1. As stated in [1], linear operations on a secret encoded by a polynomial are simple to perform. Let $p(x)$ be a polynomial of degree $k-1$ used to encode a secret *gs*. The secret *gs* is uniquely encoded by a set of k points $(x_1,y_1), \ldots, (x_k,y_k)$, where $p(x_i) = y_i$ for $i = 1, \ldots, k$. The polynomial $q_1(x)$, also of degree $k-1$, where $q_1(x_i) = y_i + \delta$, equals to $p(x) + \delta$ and encodes the secret $gs + \delta$. Similarly, the polynomial $q_2(x)$ of degree $k-1$, where $q_2(x_i) = y_i \cdot \mu$, equals to $p(x) \cdot \mu$ and encodes the secret $gs \cdot \mu$.

For example, consider the case where $p(x) = 5 + 2x + 3x^2$, a polynomial of degree 2 that encodes the global secret $gs = 5$, among a swarm of $n = 5$ members. The pairs (1,10), (2,21), (5,90), (7,166) and (10,325) are possible n secret components, where each $k = 3$ components are required to reconstruct $p(x)$ and calculate *gs*.

Incrementing the value of *gs* by $\delta = 3$ results in a new polynomial $q(x) = 8 + 2x + 3x^2$, which is represented by the following secret components: (1,13), (2,24), (5,93), (7,169) and (10,328). Similarly, when *gs* is multiplied by δ.

3.1. Polynomial based scheme – secret share of size 1

Assume that each one of the n swarm members holds a distinct secret share, which contains a single secret component $\langle (x_i,y_i) \rangle$, where $y_i = p(x_i)$ and $p(x)$ is a polynomial of degree $k-1$ encoding the swarm's global secret *gs*. According to Observation 1, adding δ to $y_1, \ldots, y_n$ results in a new polynomial $q(x)$ where $q(x) = p(x) + \delta$. Hence, increasing (decreasing) all the values y_i by δ increases (decreases) the secret by δ as well, since $q(0) = p(0) + \delta$. Also, multiplying the second coordinate by some factor δ implies the multiplication of *gs* by δ. Thus, updating the value *gs* is done by internal actions followed by a *step* input action, which specifies the arithmetic operation (multiplication or addition) and the δ by which the value of *gs* is multiplied or increased.

Based on the above, the input actions are implemented as described in Fig. 1.

3.1.1. Line-by-line code description

The code in Fig. 1 describes input actions of process *i*. Each process *i* has a secret share of a single secret component: a pair (x_i,y_i), where $y_i = p(x_i)$. 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 process and additional parameters of the input action.

- *set*: On *set*, process *i* receives a message of type *set*, indicating the *set* input action, and a secret *share*, consists of a single secret component of the form (x,y), where $y = p(x)$ (line 1). Process *i* sets x_i and y_i with the component in the received *share* (lines 2, 3), where *getX*(*share*,*j*) and *getY*(*share*,*j*) returns the x and y values, respectively, of the *j*th component in the given secret *share*.
- *step*: On *step*, process *i* receives a message of type *stp*, indicating the *step* input action, a value δ and an operation type *op* (line 4). The value δ may be negative and indicates a change in the secret share that affects the global secret. The operation *op* may be either *ADD* or *MUL*, which indicates the arithmetic addition and multiplication operations respectively.
  If *op* is *ADD*, then y_i is incremented by δ (lines 5, 6). Otherwise, the operation is *MUL* and then y_i is multiplied by δ (lines 7, 8). By Observation 1, incrementing or multiplying the global secret by δ can be done by incrementing or multiplying each value y_i of the secret component (x_i,y_i). Therefore, a step input action implies the addition or multiplication of the global secret *gs* by δ.
- *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 9), which triggers a leader election procedure (line 10). Once a leader is elected, it is responsible for collecting all the members' shares, calculating the global secret and redistributing the secret shares amongst the swarm members.

If process *i* is the leader (lines 11–19), it first listens to regain consistency reply messages sent by other swarm members. These reply messages contain the members'

[5] In the sequel we assume that a joining process reveals information equivalent to a captured process, though, if it happens that the compromising adversary is not present during the join no information is revealed.

shares, which are collected into the set of all secret components, denoted by *allSecretComponents* (line 12).

If the number of distinct secret components is fewer than k, i.e., some of the global secret components are missing and the secret cannot be reconstructed, then process i initializes a set *allSecretComponents* with the set of components returned by the method *setDefaultSecret*( ) (lines 13, 14). This method sets the values of the global secret *gs* with a predefined default value, and returns a set of n distinct secret components, which is assigned to the set *allSecretComponents*.

Having set the global secret components, process i (the leader) redistributes the secret (lines 15–17). The function *getRandomShare* returns a random share of a given size (1 in this case) out of the set *allSecretComponents* (line 16). A random share is sent to each swarm member using a *set* message to set the share of the member with the random one (line 17). After sending the shares to all members, the leader sets its own share with a random share (line 18).

Finally, after the shares are sent by the leader, the set *allSecretComponents* is initialized with an empty set, to avoid revealing the secret in case the leader is later compromised (line 19). In case process i is not the leader, it sends its share to the leader by sending a *rgn_rply* message (lines 20, 21).

- *regainConsistencyReply*: On *regainConsistencyReply*, all processes are assumed to be in a safe place without the threat of any adversary. Process i receives a message of type *rgn_rply*, which includes the secret share of the sender, identified by *srcid* (line 22). If process i is the leader, then it adds the received component to the set *allSecretComponents* (lines 23, 24). Otherwise, the message is ignored.
- *joinRequest*: An input message of type *join_rqst* indicates a request by a new process with identifier *srcid* to join the swarm (line 25). Process i sends its secret component to the joining process only if no other reply was previously sent by another swarm member. For that, it holds a variable *replyWasSent*, initialized with *false*, to indicate whether a reply of another process was sent back to the joining process (line 26). It then 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 27). During that random period of time, process i listens to join replies sent by other processes. Each reply includes a share with a single component, namely, a pair (x_j, y_j), where $y_j = p(x_j)$. If such a reply was sent, then *replyWasSent* is set with *true* (lines 28, 29). Whenever that random period of time has elapsed, if no reply was sent, then process i sends its secret component to the joining process *srcid* (lines 30, 31).

  We assume that at most one sender may succeed in sending the reply. Moreover, processes know which secret component is successfully sent. Note that the secret components can be encrypted. In this case, the *join_rqst* message may include a public key. Otherwise, we regard each join as a process captured 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 secret component for the joining process (line 32). Process i sets its share with the value of the given secret component (lines 33, 34).

3.1.2. Reconstructing the secret

Let $pr(m)$ denote the probability that a random set of m secret shares can reconstruct $p(x)$ and calculate *gs*. On regain consistency operation, the secret shares are collected from all swarm members, the global secret is reconstructed and redistributed again amongst the swarm members. Assume the number of processes in the swarm initially (or immediately after a regain consistency operation) is n. As long as there are no members which join or leave the swarm, the members hold n distinct secret components, where any fewer than k components cannot reconstruct the secret and any k or more can reconstruct it with probability 1. In this case, $pr(m) = 0$ for all $m < k$ and $pr(m) = 1$ for all $m \geqslant k$.

Assuming the number n_{lp} of leaving processes between two successive regain consistency operations is less than $n - k$, and the number n_{jp} denotes the number of joining processes, then $pr(m) \leqslant 1$, since some of the components appear more than once in the swarm. That probability is a function of n, k, n_{lp} and n_{jp}. Obviously, when all the members' components are given (as on regain consistency), that probability is 1.

3.1.3. Passive adversary

According to Shamir's (k,n)-threshold scheme, at least k distinct secret components are required to reconstruct the swarm's global secret *gs*. Therefore, in any execution in which an adversary captures at most $f < k$ processes, at least one component is missing and the secret cannot be reconstructed.

3.1.4. Active (Byzantine) adversary and error correcting

In the presence of an active adversary, which has the ability to corrupt the state of at most f swarm members, we design a scheme that is robust to faults. Having m distinct points of a polynomial $p(x)$ of degree $k - 1$, the Berlekamp-Welch decoder [16] can reconstruct the secret as long as the number of errors f is less than $(m - k + 1)/2$. If there are no *join* operations, all shares are distinct, then in the case of f errors (or corrupted state members), any set of $m > 2f + k - 1$ shares can reconstruct the secret. Note that the Berlekamp-Welch decoder refer to errors in the y_i values, but the active adversary can corrupt the x_i values as well. This fact does not affect the correctness of using the decoder, since for any value x_i, the value $y_i = p(x_i)$ may be regarded as erroneous. If there were *join* operations, then some of the shares may be duplicated and, hence, a set of $m > 2f + k - 1$ may reconstruct the secret with some positive probability.

3.2. Polynomial based scheme – secret share of size > 1

Previously we assumed that the number n_{lp} of leaving processes is at most $n - k$ between any two successive global resets of the secret. Here we assume that neither the number of leaving nor the number of joining processes is limited, but the number of processes in the swarm at any given time is at least k.

Under this assumption, consider the following scenario. Processes left and joined the swarm in a way that caused the swarm to be in a *critical state*, namely a state in which the number of processes is larger than k, there are exactly k distinct shares distributed amongst the swarm members and there is a share $share_i$ of process i, which uniquely appears in the swarm. If process i leaves the swarm, there are $k-1$ distinct shares ($k-1$ polynomial points) and the secret cannot be reconstructed. In order to increase the probability of overcoming a critical state, we use secret share which consists of s (uniformal chosen) secret components rather than a single component. That way, on the next join operation, before the process i with the unique share leaves, the probability of the joining process to get a share which includes the component in $share_i$ is s times larger. Motivated by the above scenario, we use a polynomial $p(x)$ of degree $t-1>k$ to encode the secret by Shamir's (t,t)-threshold scheme. A secret share is a tuple of s secret components (points), where $s=t/k$. In this scheme we do not use n distinct polynomial points but rather t.

The implementation of the input actions has slightly changed, as shown in Fig. 2.

3.2.1. Code description

The code in Fig. 2 describes input actions of process i, when a secret share is a tuple of s distinct secret components. A secret share $\langle(x_{i_1},y_{i_1}),(x_{i_2},y_{i_2}),\ldots,(x_{i_s},y_{i_s})\rangle$ of process i is represented by two arrays, $X_i[1\ldots s]$ and $Y_i[1\ldots s]$, where $X_i[j]$ and $Y_i[j]$ match x_{i_j} and y_{i_j} respectively. The following describes the code changes between Figs. 1 and 2.

- *set*: On *set* (line 1), process i sets its secret share with the received one (lines 2–4).
- *step*: On *step* (line 5), process i, according to the received arithmetic operation *op* (line 6), increments or multiplies each value $Y_i[j]$, $j=1\ldots s$ by δ. When done for all secret shares, this modification of the shares implies the modification of the global secret *gs* by δ.
- *regainConsistencyRequest*: On *regainConsistencyRequest*, executed in a safe place, process i receives a message of type *rgn_rqst* (line 12). Handling the request is done similarly to what is described in Fig. 1, Except the random share which is sent back to each swarm member consists of s secret components rather than 1 (lines 18–21).
- *regainConsistencyReply*: On *regainConsistencyReply*, executed in a safe place, the leader adds all the components of the received share into the set *allSecretComponents* (lines 25–27).
- *joinRequest*: An input message of type *join_rqst* indicates a request by a new process with identifier *srcid* to join the swarm (line 28). Process i waits a random period of time, during which it listens to join replies of other swarm members. If the number of distinct secret components, sent by other swarm members as join replies, is less than s, then it sends a secret component to the joining process. That secret component is randomly chosen out of the secret components in the secret share of process i.

  The join procedure is designed to restrict the shares that may be revealed by the passive adversary. For that, process i initializes an empty set *sentComponents* of secret components, which were sent by swarm members (line 29). While the number of sent secret components is less than s, process i does the following (line 30). It sets *waitingTime* with a random period of time, as specified in Section 3.2 (line 31). During that random period of time, process i listens to join replies sent by other processes, each reply includes a secret component. While listening, process i adds the sent components to its *sentComponents* set (lines 32–34). After the *waitingTime* has elapsed, if the number of distinct secret components, which were sent to the joining process *srcid*, is less than 1, then process i sends a join reply to process *srcid*. This reply message includes a secret component, randomly chosen out of the secret components in its share that are not in *sentComponents* (lines 35–37). It is done by calling the function *getRandomComponent*($X_i[1\ldots s],Y_i[1\ldots s]$,*sentComponents*), which returns a randomly chosen component out of the share that does not appear in *sentComponents*.
- *joinReply*: On *joinReply*, process i receives an input message of type *join_rply*, which contains a secret component. If the current size of its secret share is smaller than s (lines 39, 40), and the received secret component was not previously received by process i (line 43), then the received secret component is added to the share of process i (lines 44, 45).

3.2.2. Reconstructing the secret

When using t distinct secret components and share size of size $s>1$, the probability pr_m to reconstruct the secret, given m shares, randomly chosen out of the n secret shares, is in fact, the probability that all the t secret components of *gs* are present in that set of m secret shares.

Clearly, for $0\leqslant m<k$ it holds that $pr_m=0$, since at least one secret component is missing. Whereas, for $m\geqslant k$ it holds that $pr_m=[1-(1-p)^m]^t$, where p is the probability of a secret component to be chosen for a secret share. As the components are chosen with equal probability out of the t components of *gs*, it holds that $p=\frac{s}{t}=\frac{1}{k}$, assuming k divides t. The probability that a certain secret component appears in one of the m secret shares is $1-(1-p)^m$. Hence, the probability that no component is missing is $[1-(1-p)^m]^t$. Therefore, the expected number m of required secret shares is a function of m, t and k.

3.2.3. Passive adversary

According to Shamir's (t,t)-threshold scheme, at least t distinct secret components are required to reveal the swarm's global secret *gs*. Here, each process has a secret share with s distinct secret components, where $s=t/k$. Therefore, compromising at most $f<k$ processes ensures that at least one component is missing and therefore the polynomial $p(x)$ cannot be calculated.

```
set_i(⟨set, srcid, i, share⟩)
    for j = 1..s do
        X_i[j] ⟵ getX(share, j)
        Y_i[j] ⟵ getY(share, j)

step_i(⟨stp, srcid, i, op, δ⟩)
    if op == ADD
        for j = 1..s do
            Y_i[j] ⟵ Y_i[j] + δ
    else
        for j = 1..s do
            Y_i[j] ⟵ Y_i[j] * δ

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
        allSecretComponents_i ⟵ listenAll(⟨rgn_rply, i, j, share⟩)
        if size(allSecretComponents_i) < t then
            allSecretComponents_i ⟵ setDefaultSecret()
        for every process id j in the swarm do:
            new_share ⟵ getRandomShare(allSecretComponents, s)
            send(⟨set, i, j, new_share⟩)
        ⟨X_i[1..s], Y_i[1..s]⟩ ⟵ getRandomShare(allSecretComponents, s)
        allSecretComponents_i ⟵ ∅
    else
        send(⟨rgn_rply, i, leaderId, ⟨X_i[1..s], Y_i[1..s]⟩⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
        allSecretComponents_i ⟵ allSecretComponents_i ∪ {share}

joinRequest_i(⟨join_rqst, srcid, i⟩)
    sentComponents ⟵ ∅
    while |sentComponents| < s do
        waitingTime ⟵ random([1..maxWaiting(n)])
        while waitingTime not elapsed do
            listen(⟨join_rply, i, pid, component⟩)
            sentComponents ⟵ sentComponents ∪ {component}
        if |sentComponents| < s then
            random_component ⟵ getRandomComponent(X_i[1..s], Y_i[1..s], sentComponents)
            send(⟨join_rply, i, srcid, random_component⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    size ⟵ sizeof(X_i)
    if size < s then
        x ⟵ getX(component)
        y ⟵ getY(component)
        if x ∉ X_i[1..size] then
            X_i[size + 1] ⟵ x
            Y_i[size + 1] ⟵ y
```

Fig. 2. Polynomial-based solution with multiple component share, program for swarm member *i*.

3.2.4. Active adversary and error correcting

Similarly to the case of a secret share of size 1, if m members were corrupted, then at most $max\{m \cdot s, t\}$ components are corrupted. The Berlekamp-Welch decoder [16] can reconstruct $p(x)$ as long as the number of errors f is less than $(m \cdot s - k + 1)/2$ and $m \cdot s < t$.

4. Reactive *k*-secret sharing – the Chinese remainder-based scheme

Here, the representation of the global secret *gs* is based on the Chinese Remainder Theorem (CRT). Given a set of relatively prime numbers $p_1 < p_2 < \cdots < p_k, N_k = \prod_{i=1}^{t} p_i$ and an integer m such that $0 \leqslant m < N_k$, m is uniquely specified by its residues modulo $p_1 < \cdots < p_k$. If we use $n > k$ relatively prime numbers $p_1 < \cdots < p_k < p_{k+1} < \cdots < p_n$, then m can be calculated out of any k pairs (p_i, r_i), where $r_i = m \bmod p_i$.

Therefore, the global secret *gs* can be represented by a set of n such pairs (p_i, r_i), where $gs < N_k$. A *secret component* then, is a pair (p_i, r_i), where $r_i = gs \bmod p_i$ and $p_i \in P$. A *secret share* is a set of distinct secret components, as in the polynomial-based scheme. We distribute the global secret *gs* amongst the n processes in a way that k or more members may reconstruct the secret with some probability, yet any fewer than k members fail to do so.

Lemma 1. *Given a set $P = \{p_1, p_2, \ldots, p_k, \ldots, p_n\}$ of n relatively prime numbers, and $N_k = \prod_{i=1}^{k} p_i$. According to the CRT, an integer $0 \leqslant m < N_k$ is uniquely defined by a set of distinct pairs $(p_{i_1}, r_{i_1}), \ldots, (p_{i_k}, r_{i_k})$, where $r_{i_j} = m \mod p_{i_j}$ and $p_{i_j} \in P$. Then, $(m + \delta) \mod N_k$ is uniquely defined by the set $(p_{i_1}, (r_{i_1} + \delta) \mod p_{i_1}), \ldots, (p_{i_1}, (r_{i_k} + \delta) \mod p_{i_k})$.*

Proof. For every pair (p_i, r_i), where $r_i = m \mod p_i$, by definition, $\exists q$: $m = r_i + q \cdot p_i$. Note that m, $m + \delta < N_k$. Therefore, $m + \delta = r_i + \delta + q \cdot p_i$. If $r_i + \delta < p_i$, then clearly $m + \delta = (r_i + \delta) \mod p_i$. Otherwise, let $r_i' = (r_i + \delta) \mod p_i$. By definition, $\exists q' : r_i + \delta = r_i' + q' \cdot p_i$. Therefore, $m + \delta = r_i' + (q' + q) \cdot p_i . m + \delta = r_i' \mod p_i = (r_i + \delta) \mod p_i$. $\square$

Lemma 2. *Given a set $P = p_1, p_2, \ldots, p_k, \ldots, p_n$ of n relatively prime numbers, and $N_k = \prod_{i=1}^{k} p_i$. According to the CRT, an integer $0 \leqslant m < N_k$ is uniquely defined by a set of distinct pairs $(p_{i_1}, r_{i_1}), \ldots, (p_{i_k}, r_{i_k})$, where $r_{i_j} = m \mod p_{i_j}$ and $p_{i_j} \in P$. Then, $(m \cdot \delta) \mod N_k$ is uniquely defined by the set $(p_{i_1}, (r_{i_1} \cdot \delta) \mod p_{i_1}), \ldots, (p_{i_1}, (r_{i_k} \cdot \delta) \mod p_{i_k})$.*

Proof. For every pair (p_i, r_i), where $r_i = m \mod p_i$, by definition, $\exists q$: $m = r_i + q \cdot p_i$. Note that m, $m \cdot \delta < N_k$. Therefore, $m \cdot \delta = r_i \cdot \delta + q \cdot \delta \cdot p_i$. If $r_i \cdot \delta < p_i$, then clearly $m \cdot \delta = r_i \cdot \delta \mod p_i$. Otherwise, let $r_i' = (r_i \cdot \delta) \mod p_i$. By definition, $\exists q' : r_i \cdot \delta = r_i' + q' \cdot p_i$. Therefore, $m \cdot \delta = r_i' + (q' + q) \cdot p_i . m \cdot \delta = r_i' \mod p_i = r_i \cdot \delta \mod p_i$. $\square$

Note that if $r_i = gs \mod p_i$ for every $p_i \in P$, then $r_i + \delta = (gs + \delta) \mod p_i$ for every $p_i \in P$. Therefore, adding δ to gs can be done by adding δ to each residue r_i modulo p_i. Similarly, multiplying gs by some δ, is done by multiplying each residue r_i by δ modulo p_i.

For example, let $P = \{2,3,5,7\}$ $(p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 7)$, $k = 3$, $n = 4$ and $gs = 0$. Also, $0 \leqslant gs < N_k$, where $N_k = 2 \cdot 3 \cdot 5 = 30$. Assume gs is initially zero. Hence, the CRT-representation of $gs = 0$, using P, is $\langle(2,0), (3,0), (5,0), (7,0)\rangle$. After incrementing the value of gs by $\delta = 1$ it holds that the CRT-representation of $gs = 1$ is $\langle(2,1), (3,1), (5,1), (7,1)\rangle$. Incrementing gs by $\delta = 5$ again, results in $gs = 6$, represented by $\langle(2,0), (3,0), (5,1), (7,6)\rangle$. Incrementing gs by $\delta = 4$, results in $gs = 10$, represented by $\langle(2,0), (3,1), (5,0), (7,3)\rangle$. Multiplying gs by 2 results in $gs = 20$ represented by $\langle(2,0), (3,2), (5,0), (7,6)\rangle$.

4.1. Chinese remainder-based scheme – secret share of size 1

Assume a set of n relatively prime numbers $P = \{p_1, \ldots, p_k, \ldots, p_n\}$, where $p_1 < p_2 < \cdots < p_k < \cdots < p_n$. The product $\prod_{i=1}^{k} p_i$ is denoted by N_k and 0*les* $gs < N_k$. According to the CRT, any k pairs (p_i, r_i) where $p_i \in P$ can reconstruct gs. Yet, any fewer than k pairs cannot reconstruct it. Therefore, in this scheme, each swarm member holds a secret share of one secret component (p_i, r_i).

In this case, the implementation of the input actions is very similar to that of the polynomial-based scheme, Only here we use a secret component of the form (p_i, r_i), where $r_i = gs \mod p_i$, rather than (x_i, y_i), where $y_i = p(x)$. Also, this solution does not support arithmetic multiplication of gs.

The implementation is described in Fig. 3.

Reconstructing gs is similar to the polynomial-based scheme.

4.1.1. Passive adversary

Similarly to the polynomial-based scheme, at least k distinct secret components are required to reveal the swarm's global secret gs. Therefore, in any execution in which an adversary compromises at most $f < k$ processes, it cannot reveal the secret. Moreover, we assume that there is a lower bound $p_0 \notin P$ on the relatively prime numbers in P such that $p_0 < p_1 < p_2 < \cdots < p_n$. The use of $n > k$ relatively prime numbers, where only k are required naturally yields an error correcting code [9]. An adversary which compromises at most $f < k$ swarm members is missing at least one pair, denoted by (p_j, r_j). Had it known the prime, the probability of the adversary to guess the secret was uniform over the values $0, 1, \ldots, p_j - 1$, which is bounded by $\frac{1}{p_0}$.

4.1.2. Active (Byzantine) adversary and error correcting

In the presence of an active adversary, which has the ability to corrupt the state of at most $f < k$ swarm members, we design a scheme that is robust to faults. The global secret $gs < N_k$ is uniquely specified by its residue modulo $p_1 < \cdots < p_k$. Our scheme uses $n - k$ redundant primes $p_{k+1} < \cdots < p_n$ for representing gs. Note that on the presence of errors, the primes may also be faulty, but similar to the polynomial-based scheme, an error in a prime p_i may be regarded as an error in the residue r_i. The only problem is that the erroneous prime p_i may not be relatively prime with $p_j \in P$, where $j \neq i$. For that, we assume that in the presence of an active adversary the set $P = \{p_1, p_2, \ldots, p_n\}$ is a set of n prime numbers (in particular, a set of n relatively prime numbers). On regaining consistency, any pair (p_i, r_i) which is received by the leader is discarded if p_i is not prime. Under this assumption, we can update the *regainConsistency* input action, so that the processes first agree on $\mathscr{P}$ by a simple majority function. Then, any residue paired with a prime in P may be chosen. Then, they can use Mandelbaum's technique [13] in order to correct the errors and reconstruct gs. In this case, the number of Byzantine values or errors, modeled by f, is required to be less than the majority and less than $(n - k)/2$. By this technique, at most $n - k$ errors can be detected and $(n - k)/2$ can be corrected.

4.2. Chinese remainder-based scheme – secret share of size >1

Motivated by the need to decrease the probability of a *critical state*, as described in Section 3.2, we design a scheme in which each secret share is a tuple of s secret components rather than a single component.

Here, we use a set $P = \{p_1 < p_2 < \cdots < p_t\}$ of $t > k$ relatively prime numbers $\{p_1 < p_2 < \cdots < p_t\}$. The global secret gs is bounded by $N_t = \prod_{i=1}^{t}$ and a secret share is a tuple of s secret components (points), where $s = t/k$. In this solution we do not use n distinct components but rather t.

The implementation of the input actions is similar to the polynomial-based scheme described in Section 3.2, see Fig. 4. A secret share $\langle(p_{i_1}, r_{i_1}), \ldots, (p_{i_s}, r_{i_s})\rangle$ is represented by two arrays $P_i[1 \ldots s]$ and $R_i[1 \ldots s]$ of process i, where $p_{i_j} = P_i[j]$ and $r_{i_j} = R_i[j]$.

```
set_i(⟨set, srcid, i, share⟩)
    p_i ⟵ getP(share, 1)
    r_i ⟵ getR(share, 1)

step_i(⟨stp, srcid, i, op, δ⟩)
    if op == ADD
        r_i ⟵ (r_i + δ) mod p_i
    else
        r_i ⟵ (r_i * δ) mod p_i

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
        allSecretComponents_i ⟵ listenAll(⟨rgn_rply, j, i, share⟩)
        if size(allSecretComponents_i) < k then
            allSecretComponents_i ⟵ setDefaultSecret()
        for every process id j in the swarm do:
            new_share ⟵ getRandomShare(allSecretComponents, 1)
            send(⟨set, i, j, new_share⟩)
        ⟨(p_i, r_i)⟩ ⟵ getRandomShare(allSecretComponents, 1)
        allSecretComponents_i ⟵ ∅
    else
        send(⟨rgn_rply, i, leaderId, ⟨(p_i, r_i)⟩⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
        allSecretComponents_i ⟵ allSecretComponents_i ∪ share

joinRequest_i(⟨join_rqst, srcid, i⟩)
    replyWasSent ⟵ false
    waitingTime ⟵ random([1..maxWaiting(n)])
    while waitingTime not elapsed do
        replyWasSent ⟵ listen(⟨join_rply, j, srcid, component⟩)
    if replyWasSent = false then
        send(⟨join_rply, i, srcid, (p_i, r_i)⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    p_i ⟵ getP(component)
    r_i ⟵ getR(compoent)
```

Fig. 3. Chinese remainder-based solution with single component share, program for swarm member *i*.

4.2.1. Reconstructing the secret

As in the polynomial-based scheme, the probability pr_m that a random set of m random secret shares can reconstruct the secret gs is 0 for $m < k$ and $[1-(1-p)^m]^t$ for $m \geqslant t$.

4.2.2. Passive adversary

According to the CRT, at least t distinct secret components are required to reconstruct gs. Similarly to the polynomial-based scheme, if an adversary compromises at most $f < k$ swarm members then it reveals at most $(k-1)s = (k-1)t/k$ distinct secret components. Namely, at least one secret component is missing.

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

Proof. In any execution in which the adversary captures at most $f < k$ processes, there is at least one missing secret component, say (p,r). Assume that the adversary knows the missing prime p. In this case, the adversary has p possible values $\{0,1,2,\ldots,p-1\}$ for r, out of which only one is the correct value. Therefore, in the case of knowing the missing prime p, the probability to reveal the secret is $\frac{1}{p} < \frac{1}{p_{min}}$.

Moreover, when the adversary does not know the value of p, the probability to guess the secret is even less than $\frac{1}{p_{min}}$. $\square$

4.2.3. Active adversary and error correcting

We now turn to considering the case of an active (Byzantine) adversary, in which some errors take place, such as input not received by all swarm members. Similarly to the Chinese remainder-based scheme with a secret share of

```
set_i(⟨set, srcid, i, share⟩)
    for j = 1..s do
        P_i[j] ⟵ getP(share, j)
        R_i[j] ⟵ getR(share, j)

step_i(⟨stp, srcid, i, op, δ⟩)
    if op == ADD
        for j = 1..s do
            R_i[j] ⟵ (R_i[j] + δ) mod P_i[j]
    else
        for j = 1..s do
            R_i[j] ⟵ (R_i[j] * δ) mod P_i[j]

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
        allSecretComponents_i ⟵ listenAll(⟨rgn_rply, i, j, share⟩)
        if size(allSecretComponents_i) < t then
            allSecretComponents_i ⟵ setDefaultSecret()
        for every process id j in the swarm do:
            new_share ⟵ getRandomShare(allSecretComponents, s)
            send(⟨set, i, j, new_share⟩)
        ⟨P_i[1..s], R_i[1..s]⟩ ⟵ getRandomShare(allSecretComponents, s)
        allSecretComponents_i ⟵ ∅
    else
        send(⟨rgn_rply, i, leaderId, ⟨P_i[1..s], R_i[1..s]⟩⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
        allSecretComponents_i ⟵ allSecretComponents_i ∪ {share}

joinRequest_i(⟨join_rqst, srcid, i⟩)
    sentComponents ⟵ ∅
    while |sentComponents| < s do
        waitingTime ⟵ random([1..maxWaiting(n)])
        while waitingTime not elapsed do
            listen(⟨join_rply, i, pid, component⟩)
            sentComponents ⟵ sentComponents ∪ {component}
        if |sentComponents| < s then
            random_componentgetRandomComponent(P_i[1..s], R_i[1..s])
            send(⟨join_rply, i, srcid, random_component⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    size ⟵ sizeof(X_i)
    if size < s then
        x ⟵ getP(component)
        y ⟵ getR(component)
        if x ∉ P_i[1..size] then
            P_i[size + 1] ⟵ x
            R_i[size + 1] ⟵ y
```

Fig. 4. Chinese remainder-based solution with multiple component share. Program for swarm member *i*.

size 1, Mandelbaum's technique can be used to cope with errors.

5. Reactive *k*-secret sharing – Vandermonde matrix-based scheme

Here we consider a global secret *gs*, which is updated using bitwise-*xor* operations for the limited case in which $t = k = n - 1$. A random Vandermonde matrix $M^{n \times k}$ of n rows and $k = n - 1$ columns is used to encode the global secret *gs*, where $0 \leqslant gs < 2^k$. The matrix M, known to all swarm members, is over a binary field and any k rows out of n are independent. Note that M can be created by using any $k \times k$ matrix with linearly independent rows and an additional row that is thier sum. The global secret is, in fact, a binary vector *gs* of size k.

Given a binary vector v of m bits, let $v[j]$ denote the jth bit of v, where $j = 1, \ldots, m$. Similarly, the jth row of a matrix M is denoted by the vector $M[j]$. A *secret component* is a pair (j, b_j), where b_j is the *xor* of all the bits $gs[l]$, where $M[j][l] = 1$ and $M[j][l]$ denotes the lth bit in the jth row of M.

For example, let $n = 4$, $k = 3$ and $gs = (101)$, i.e., the value of the global secret is 5.

Let $M^{n\times k}$ be the following matrix, for which every three rows are linearly independent:

$$M = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Note that a row in M that include a single bit with value 1 reveals a bit of gs. Thus, we suggest to *xor* the secret with a random string and set the shares as shares of the resulting masked secret. Then, on regain consistency, when the swarm is in a safe land, the random string may be used again to reconstruct the secret. The pairs (1,1), (2,0), (3,0) and (4,1) are secret components of the global secret gs. The first row $M[1]$ of M is a binary vector that defines which bits of the vector gs are *xor*ed to compute b_1. In this case, the first and second bits of gs are *xor*ed and the result is $b_1 = 1$.

Note that given a binary vector v of k bits, updating the global secret $gs = gs \oplus v$ is done by updating each secret component (j, b_j) to be $\left(j, b'_j\right)$, where b'_j is the value b_j *xor*ed with all the bits $v[l]$, where $M[j][l] = 1$.

For example, for the specified gs and M, let $v = (010)$. The above secret components are updated to be the following: $(1,b'_1) = (1, 1\oplus 0 \oplus 1) = (1,0), (2,b'_2) = (2, 0\oplus 0 \oplus 0) = (2,0), (3,b'_3) = (3, 0\oplus 1) = (3,1)$ and $(4,b'_4) = (4, 1\oplus 0) = (4,1)$.

Assume that each one of the n swarm members holds a distinct secret share, which contains a single secret component $\langle (j, b_j)\rangle$, as specified, where $M^{n\times k}$ encodes the global secret gs, and $0 \leqslant gs < 2^k$.

The input actions are implemented as described in Fig. 5. A secret share with single secret component $\langle (j, b_j)\rangle$ of process i is represented by an index $row_i = j$ and a bit $bit_i = b_j$.

5.0.4. Reconstructing the secret

Assume the number of processes in the swarm initially (or immediately after a global reset) is n. As long as there are no members which join or leave the swarm, the members hold n distinct secret components. Any $k = n - 1$ or more components can reconstruct the secret, as they can create a $k \times k$ matrix of k independent rows which uniquely encodes the binary vector gs. In this case, the probability pr_m to reconstruct the secret out of m secret components is 0 for $m < k$ and 1 for $m \geqslant k$.

5.0.5. Passive adversary

At least k rows of M are required to reconstruct the binary vector gs, hence, at least k distinct secret components are required to reconstruct gs. Therefore, in any execution in which an adversary captures at most $f < k$ processes, at least one component is missing and hence gs cannot be reconstructed.

6. Virtual automaton

We would like the swarm members to implement a virtual automaton where the state is unknown. Thus, if at most f 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 [12] 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. The set of local controlled actions is denoted by $local(A) = out(A) \cup int(A)$.
- A set $states(A)$ of states.
- A non-empty set $start(A) \subseteq states(A)$ of initial states.
- A transition relation $steps(A) : states(A) \times acts(A) \to states(A)$, 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 as specified. The swarm's *global state* is the current state in the execution of A and is, in fact, The swarms actual global secret. A secret component is a state $s \in states(A)$ and a secret share is simply a tuple of m states $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_m}\rangle$ of m distinct states, where $s_{i_j} \in states(A)$ for all $j = 1\ldots m$ 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 state tuples ($T \leqslant n$). If there is more than one such state, 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 $j = 1, \ldots, m$. The swarm's *global output* is defined to be the result of the output action which appears in at least threshold T out of n members' output.

We assume the existence of devices (sensors, for example) which receive 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 $f < T$ 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 state tuples of the compromised processes and assume that the most common state, i.e., appears the most frequently in the compromised state tuples, is most likely to be the global state of the swarm.

Consider the case in which $f = 1$ and $T = \lfloor n/2 + 1 \rfloor$. The secret share of process i is denoted by $state\_tuple_i$. If the $|state\_tuple_i| = 1$, i.e., the secret share includes a single state, then an adversary which compromises process i

```
set_i(⟨set, srcid, i, share⟩)
    row_i ⟵ getRow(share, 1)
    b_i ⟵ getBit(share, 1)

step_i(⟨stp, srcid, i, v⟩)
    b_i ⟵ b_i ⊕ v[j_1] ⊕ ... ⊕ v[j_l], where M[row_i][j_1] = ... = M[row_i][j_l] = 1

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
        allSecretComponents_i ⟵ listenAll(⟨rgn_rply, j, i, share⟩)
        if size(allSecretComponents_i) < k then
            allSecretComponents_i ⟵ setDefaultSecret()
        for every process id j in the swarm do:
            new_share ⟵ getRandomShare(allSecretComponents, 1)
            send(⟨set, i, j, new_share⟩)
        ⟨(row_i, b_i)⟩ ⟵ getRandomShare(allSecretComponents, 1)
        allSecretComponents_i ⟵ ∅
    else
        send(⟨rgn_rply, i, leaderId, ⟨(row_i, b_i)⟩⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
        allSecretComponents_i ⟵ allSecretComponents_i ∪ share

joinRequest_i(⟨join_rqst, srcid, i⟩)
    sentComponents ⟵ ∅
    while |sentComponents| < 1 do
        waitingTime ⟵ random([1..maxWaiting(n)])
        while waitingTime not elapsed do
            listen(⟨join_rply, j, srcid, component⟩)
            sentComponents ⟵ sentComponents ∪ {component}
        if |sentComponents| < 1 then
            send(⟨join_rply, i, srcid, ⟨(row_i, b_i)⟨⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    row_i ⟵ getRow(component)
    b_i ⟵ getBit(compoent)
```

Fig. 5. Vandermonde matrix-based solution with single component share. Program for swarm member *i*.

knows the secret share $state\_tuple_i = \langle s_{i_1} \rangle$. 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 $|state\_tuple_i| = 2$, then an adversary which compromises process i, reveals the secret share $state\_tuple_i = \langle s_{i_1}, s_{i_2} \rangle$. The probability that either one 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 of the two states is most likely to be the swarm's global state, the only option for an 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 that case. Generally, if $|state\_tuple_i| = m$, 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 $state\_tuple_i$ increases, the probability to reveal the swarm's global state decreases.

The input actions are implemented as follows:

- *set* $(\langle s_{i_1}, \ldots, s_{i_m} \rangle)$: Sets the secret share $state\_tuple_i$ with the given share (tuple). The tuples are distributed in a way that at least $T + f + n_{lp}$ of them contain the swarm's global state immediately after a global reset of the secret or a regain consistency execution. Thus, even if f shares are corrupted and n_{lp} processes have left the swarm, the swarm threshold is respected. Moreover, in order to ensure the uniqueness of the global state in the presence of corruptions and joins, any other state should have fewer than $T - f$ replicas.
- *step*(δ): Simulates a step of the automaton for each of the states in the secret share. By the end of the simulation, each process has an updated output tuple. Here, δ is any possible input of the simulated automaton. Note that it is possible that transition reduces the number of distinct states in the tuple, in such a case, the process replaces copies of a state that already exists by a distinct state that is randomly chosen out of the states not in $state\_tuple_i$.
- *regain consistency*: Ensures that there are at least $T + f + n_{lp}$ members i, where $state\_tuple_i$ includes the swarm's global state and any other state has fewer than $T - f$ replicas.
- *join*: A process joins the swarm and constructs its secret share by collecting states from other processes. These

shares are randomly chosen out of the secret shares of these processes.

Note that the scheme benefits from smooth joins, since the number *f* that includes the join operations is taken into 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 heart beats) 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.

The code in Fig. 6 describes input actions of process *i*, when a secret share is a tuple of *m* distinct states in *states*(*A*) and at most one state is the swarm's global state. A secret share $\langle s_{i_1}, s_{i_2}, \ldots, s_{i_m} \rangle$ of process *i* is denoted by an array *state_tuple*$_i$[1…*m*], where *state_tuple*$_i$[*j*] matches s_{i_j} for all $j = 1\ldots m$. Similarly, an output tuple $\langle o_{i_1}, o_{i_2}, \ldots, o_{i_m} \rangle$ of process *i* is represented by an array *output*$_i$[1…*m*], where *output*$_i$[*j*] matches o_{i_j} for all $j = 1\ldots m$.

6.1. Code description

- *set*: On input action *set*, process *i* receives a message of type *set* and a secret share of *m* distinct states in

```
set_i(⟨set, srcid, i, share⟩)
    for j = 1..m do
            state_tuple_i[j] ⟵ getState(share, j)

step_i(⟨stp, srcid, i, δ⟩)
    for j = 1..m do
            (state_i[j], output_i[j]) ⟵ follow the transaction in steps(A) for state_tuple_i[j] and δ
    if exists j1 ≠ j2 such that states_i[j1] = states_i[j2]
            state_tuple_i[j2] ⟵ getRandomState(states(A)\states_i)
    executeOutputActions(output_acts)

regainConsistencyRequest_i(⟨rgn_rqst, srcid, i⟩)
    leaderId ⟵ leaderElection()
    if leaderId = i then
            allStateTuples ⟵ listenAll(⟨rgn_rply, i, j, share⟩)
            candidates ⟵ mostPopularStates(allStateTuples)
            if |candidates| == 1 then
                  globalState ⟵ first(candidates)
            else
                  globalState ⟵ setDefaultState
            distributeStateTuples(globalState)
            allStateTuples ⟵ ∅
            delete candidates
    else
            send(⟨rgn_rply, i, leaderId, state_tuple_i, ⟩)

regainConsistencyReply_i(⟨rgn_rply, srcid, i, share⟩)
    if leaderId = i then
            allSecretComponents_i ⟵ allSecretComponents_i ∪ {share}

joinRequest_i(⟨join_rqst, srcid, i⟩)
    sentComponents ⟵ ∅
    while |sentComponents| < m do
            waitingTime ⟵ random([1..maxWaiting(n)])
            while waitingTime not elapsed do
                  listen(⟨join_rply, i, pid, component⟩)
                  sentComponents ⟵ sentComponents ∪ {component}
            if |sentComponents| < m then
                  random_component ⟵ getRandomComponent(state_tuple_i, sentComponents)
                  send(⟨join_rply, i, srcid, random_component⟩)

joinReply_i(⟨join_rply, srcid, i, component⟩)
    size ⟵ state_tuple_i
    if size < m then
            s ⟵ getState(component)
            if s ∉ state_tuple_i then
                  state_tuple_i[size + 1] ⟵ s
```

Fig. 6. Virtual automaton, program for swarm member *i*.

states(A) (line 1). It then sets its share $state\_tuple_i$ with the received share, using the function *getState*(*share*,*j*) which returns the jth state in the given share (lines 2, 3).

- *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 4). For every state $state\_tuple_i[j]$, process i simulates the automaton A by executing a single transaction on $state\_tuple_i[j]$ and the input δ (lines 5, 6). As a result, the state $state\_tuple_i[j]$ and the output $output_i[j]$ are updated with the new state and output action according to the executed transition. If there exists $j_1 \neq j_2$, where $state\_tuple_i[j_1]$ and $state\_tuple_i[j_2]$ were updated with the same state, say s. Then, $state\_tuple_i[j_2]$ is set with a random state, not in $state\_tuple_i$ (lines 7, 8). Finally, process i executes the output actions in $output_i$ (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 secret components states in *states* (A) by listening to join replies of other swarm members (line 13). It then, executes the method*mostPopularStates*( ) in order to find the candidates to be the swarm's global state (line 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 secret share $state\_tuple_i$ 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, then it adds the received tuple to the set $allStateTuples_i$ of already received tuples (lines 25, 26). 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). This operation is done much like the polynomial-based solution described in Fig. 6.
- *joinReply*: On input action *joinReply* process i receives a message of type *join_rply* from the process identified by *srcid* (line 37). Similarly to the polynomial-based solution, it collects m distinct secret components to compose a secret share.

7. Conclusions

We have presented four schemes for reactive k-secret sharing that require no internal communication to perform a transition.

The first three solutions maybe combined as part of the reactive automaton to define a share of the state, for example to enable an output of the automaton whenever a share value of the secret is prime. Thus the operator of the swarm may control the output of each process by manipulating the secret value, e.g., making sure that secret shares are never prime, until a sufficient number and combination of events occurs. And last, the similarity in usage of Mandelbaum and Berlekamp-Welch techniques may call for arithmetic generalization of the concepts.

References

[1] M. Ben-Or, S. Goldwasser, A. Wigderson, Completeness theorems for non-cryptographic fault-tolerant distributed computation, in: Proc. of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, 1988, pp. 1–10.
[2] S. Dolev, S. Gilbert, L. Lahiani, N. Lynch, T. Nolte, Virtual stationary automata for mobile networks, in: Proc. of the 2005 International Conference On Principles Of Distributed Systems, (OPODIS), LNCS 3974, 2005 (Also invited paper in Forty-Third Annual Allerton Conference on Communication, Control, and Computing; Also, Brief announcement in Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing, (PODC 2005), 2005, pp. 323; Technical Report MIT-LCS-TR-979, Massachusetts Institute of Technology, 2005).
[3] S. Dolev, S. Gilbert, A.N. Lynch, E. Schiller, A. Shvartsman, J. Welch, Virtual mobile nodes for mobile ad hoc networks, in: International Conference on Principles of DIStributed Computing (DISC 2004), 2004, pp. 230–244 (Also Brief announcement in Proc. of the 23th Annual ACM Symp. on Principles of Distributed Computing (PODC 2004), 2004).
[4] S. Dolev, S. Gilbert, N.A. Lynch, A. Shvartsman, J. Welch, GeoQuorum: implementing atomic memory in ad hoc networks, Distributed Computing 18 (2) (2005) 125–155. special issue of selected paper from DISC 2003.
[5] S. Dolev, S. Gilbert, E. Schiller, A. Shvartsman, J. Welch, Autonomous virtual mobile nodes, in: Third ACM/SIGMOBILE Workshop on Foundations of Mobile Computing (DIALM/POMC), 2005, pp. 62–69 (Brief Announcement in Proc. of the 17th International Conference on Parallelism in Algorithms and Architectures (SPAA 2005), 2005, pp. 215; Technical Report MIT-LCS-TR-992, Massachusetts Institute of Technology, 2005).
[6] S. Dolev, T. Herman, L. Lahiani, Polygonal broadcast, secret maturity and the firing sensors, Ad Hoc Networks Journal 4 (4) (2006) 447–486.
[7] S. Dolev, L. Lahiani, N. Lynch, T. Nolte, Self-stabilizing mobile location management and message routing, in: Proc. of the 7th International Symposium on Self-Stabilizing Systems (SSS 2005), LNCS 3764, 2005, pp. 96–112 (Also Technical Report MIT-LCS-TR-999, Massachusetts Institute of Technology, 2005).
[8] S. Dolev, L. Lahiani, M. Yung, Secret swarm unit – reactive k-secret sharing, in: Proc. of the 8th International Conference on Cryptology, LNCS 4859, (INDOCRYPT 2007), December 2007, pp. 123–137 (Technical Report #12 2007, Department of Computer Science, Ben-Gurion University, September 2007).
[9] O. Goldrich, D. Ron, M. Sudan, Chinese remaindering with errors, in: Proc. of 31st STOC, ACM, 1999.
[10] E. Kivelevich, P. Gurfil, UAV flock taxonomy and mission execution performance, in: Proc. of the 45th Israeli Conference on Aerospace Sciences, 2005.

[11] J. Kilian, E. Kushilevitz, S. Micali, R. Ostrovsky, Reducibility and completeness in multi-party private computations, in: Proceedings of Thirty-fifth Annual IEEE Symposium on the Foundations of Computer Science (FOCS-94), 2000, pp. 1189–1208 (Journal version in SIAM J. Comput. 29(4)).
[12] N. Lynch, M. Tuttle, An introduction to input/output automata, in: Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, vol. 2(3), September 1989, pp. 219–246 (Also Tech. Memo MIT/LCS/TM-373).
[13] D. Mandelbaum, On a class of arithmetic and a decoding algorithm, IEEE Transactions on Information Theory 21 (1) (1976) 85–88.
[14] A. Shamir, How to share a secret, CACM 22 (11) (1979) 612–613.
[15] M. Weiser, The Computer for the 21th Century, Scientific American, 1991.
[16] L. Welch, E.R. Berlekamp, Error Correcting for Algebraic Block Codes, U.S. Patent 4633470, September 1983.

Shlomi Dolev received his B.Sc. in Engineering and B.A. in Computer Science in 1984 and 1985, and his M.Sc. and D.Sc. in computer Science in 1990 and 1992 from the Technion Israel Institute of Technology. From 1992 to 1995 he was at Texas A&M University postdoc of Jennifer Welch. In 1995 he joined the Department of Mathematics and Computer Science at Ben-Gurion University where he is now an full professor. He was a visiting researcher/professor at MIT, DIMACS, and LRI, for several periods during summers. He is the author of the book "self-stabilization" published by the MIT Press. He published two hundrends journal and conference scientific articles, and patents. He served in the program committee of more than sixty conferences including: the ACM Symposium on Principles of Distributed Computing, and the International Symposium on DIStributed Computing. He is an associate editor of the IEEE Transactions on Computers, the AIAA Journal of Aerospace Computing, Information and Communication and a guest editor of the Distributed Computing Journal and the Theoretical Computer Science Journal. His research grants include IBM faculty awards, Intel academic grants, and the NSF. He is the founding chair of the computer science department at Ben-Gurion university, where he now holds the Rita Altura trust chair in computer science. His current research interests include distributed computing, distributed systems, security and cryptography and communication networks; in particular the self-stabilization property of such systems. Recently, he is involved in optical computing research

Limor Lahiani received her Ph.D. in Computer Science degree from Ben-Gurion University of the Negev in 2008 and is with Microsoft since then. Her research interests include distributed algorithms, communication networks and algorithm for communication in sensor networks. Limor received her B.Sc. and M.Sc. in mathematics and computer science from the Ben-Gurion University of the Negev in 2002 and 2004, repectively.

Moti Yung is a Research Scientist with Google. He is also an Adjunct Senior Research Faculty in the computer science department of Columbia University. Before that, he was a technology consultant to leading companies and governments, a member of RSA Labs, a Chief Scientist of CertCo Inc. (originally, Bankers Trust Electronic Commerce), and a member of IBM Research. His main current research interests are in the areas of security, cryptography, and privacy.