Secret Sharing Krohn-Rhodes: Private and Perennial Distributed Computation*

Shlomi Dolev[1] Juan Garay[2] Niv Gilboa[3] Vladimir Kolesnikov[4]
[1]Dept. of Computer Science, Ben-Gurion University [2]AT&T Labs – Research
[3]Dept. of Computer Science, Ben-Gurion University and Deutsche Telekom Laboratories [4]Bell Laboratories
dolev@cs.bgu.ac.il garay@research.att.com niv.gilboa@gmail.com kolesnikov@research.bell-labs.com

Abstract: In this paper we consider the problem of n agents wishing to perform a given computation on common inputs in a privacy preserving manner, in the sense that even if the entire memory contents of some of them are exposed, no information is revealed about the state of the computation, and where there is no *a priori* bound on the number of inputs. The problem has received ample attention recently in the context of *swarm computing* and Unmanned Aerial Vehicles (UAV) that collaborate in a common mission, and schemes have been proposed that achieve this notion of privacy for arbitrary computations, at the expense of one round of communication per input among the n agents.
In this work we show how to *avoid communication altogether* during the course of the computation, with the trade-off of computing a smaller class of functions, namely, those carried out by finite state automata. Our scheme, which is based on a novel combination of secret-sharing techniques and the Krohn-Rhodes decomposition of finite state automata, achieves the above goal in an information-theoretically secure manner, and, furthermore, does not require randomness during its execution.

Keywords: private computation, information-theoretic security, finite state automata, Krohn-Rhodes decomposition.

1 Introduction

There is great interest in pervasive *ad hoc* and "swarm" computing [18], particularly in swarming Unmanned Aerial Vehicles (UAV) that collaborate in a common mission (e.g., [4, 10] and references therein). Hiding the state of the swarm from (a subset of) the swarm participants while keeping and updating the global swarm state, due to a possibly *unbounded* sequence of inputs, is an important task (e.g., [4, 6]). In this work we make progress in this area, by showing the feasibility of non-interactive private distributed computation on such unbounded input sequences.

Recently, Dolev *et al.* [4] presented schemes that support infinite private computation by implementing a distributed version of a so-called *strongly oblivious* universal Turing machine (TM)[1]. However, this is done at the expense of one round of communication per received input amongst the participants. In contrast, in this work we show how to avoid communication altogether during the course of the computation, with the trade-off of computing a smaller class of functions.

1.1 Our setting and goal

Specifically, we consider a distributed computation setting in which a party, which we refer to as *the dealer*, has a finite state automaton (FSA) $\mathcal{A}$ which accepts an (*a priori* unbounded) stream of inputs $x_1, x_2, \ldots$ received from an external source. We are interested in situations in which the dealer cannot perform the required computation, but instead delegates the responsibility to agents $P_1, \ldots, P_n$. Each of the

*Work partially supported by DIMACS, FRONTS EU Project, US Air Force European Office of Aerospace and Development, grant #FA8655-09-1-3016, the Rita Altura trust chair in computer sciences, Deutsche Telekom Laboratories at Ben-Gurion University of the Negev, and Lynne and William Frankel Center for Computer Sciences.

[1]An *oblivious* TM moves the tape heads through the same sequence of cells; a *strongly oblivious* TM is a Turing machine in which the movement of tape heads is a function only of the cell indices that the heads point to. Not every oblivious TM is strongly oblivious, since the movement of the tape heads may be a function of time, not only of space.

agents receives all the inputs destined to $\mathcal{A}$ during its execution. The agents execute their distributed implementation of $\mathcal{A}$ (without communication!) and, at a given signal from the dealer, terminate the execution, compute the current state of $\mathcal{A}$, and return it as output.

Furthermore, there is an additional entity, called *the adversary Adv*, who is able to adaptively "corrupt" a subset of the agents (i.e., inspect their internal state) during the execution phase, up to a threshold[2] $t < n$, and our objective is to ensure that the agents' computation is as private as possible. We do not aim to maintain the privacy of the automaton $\mathcal{A}$; however, we wish to protect the secrecy of the state of $\mathcal{A}$ and the inputs' history. We note that *Adv* may have external information about the computation, such as partial inputs or length of the input sequence, state information, etc. This auxiliary information, together with the knowledge of $\mathcal{A}$, may exclude the protection of certain configurations, or even fully determine $\mathcal{A}$'s state. We stress that this cannot be avoided in any implementation, and we do not consider this an insecurity. Thus, our goal is to prevent the leakage or derivation by *Adv* of any knowledge from seeing the execution traces which *Adv* did not already possess.

1.2 Our approach

We present a scheme that achieves the above goal in an information-theoretically secure manner (i.e., there are no bounds imposed on *Adv*'s computational power), and does not require randomness during the execution of $\mathcal{A}$. Our scheme is based on a novel combination of secret-sharing techniques [17] and the Krohn-Rhodes decomposition of finite automata [12, 13]. Informally, Krohn-Rhodes theory states that any finite state automaton can be emulated by a combination (*cascade product*—see Section 2) of permutation automata and flip-flop automata. (A permutation automaton is any automaton such that each of its possible input symbols induces a permutation of the automaton's states.) The computation complexity per each received input symbol, and the storage complexity required by our scheme are a function only of (the decomposition of) $\mathcal{A}$, and not of the number of symbols processed. A trade-off for this is that, depending on $\mathcal{A}$, the number of components of its Krohn-Rhodes decomposition might be exponential in its number of states.

[2]We note that more general access structures may be naturally employed with our constructions; see Section 2.

We note, however, that for many interesting and relevant automata, there is a small Krohn-Rhodes decomposition. Section 4.4 presents an example of such an automata family with a small Krohn-Rhodes representation.

For ease of exposition, in this submission we concentrate on the case of passive corruptions—i.e., *Adv* is considered "honest but curious." However, since our construction does not require communication among parties at the time when corruptions are allowed, it can be readily strengthened to handle active corruptions by employing secret-sharing schemes (e.g., *unverified* secret sharing [5, 16]) that are robust against disruptive behavior, and suitable for our scenario.

As noted earlier, swarms and sensor networks (e.g., [4, 6] are areas that can potentially benefit from our scheme. Another area of great current interest where user privacy is critical is that of outsourcing computation and storage to the "cloud." Yet, a big challenge in making the shift in computing to the cloud infrastructure is finding a way to ensure the privacy of the computation. One possible approach is for the users to run the program distributively on several computing clouds in such a way that even if some of them collude and exchange information they still will not be able to learn the program and/or the data they use for the computation. Furthermore, the type of computation may be of a "never-ending" nature, such as ongoing sequence of tasks performed by an operating system; the output of a given task or state of an on-going system can then be revealed by receiving information from all or a sufficient number of cloud suppliers participating in the computation—very much like a terminal client is used to interface with remote computers. Our work also addresses this scenario.

1.3 Related work

Reactive k-secret sharing with no communication among agents participating in a swarm has been suggested in [6]. Several solutions that are able to withstand limited memory corruptions were presented, some of them based on the linearity of secret sharing, supporting addition and multiplication by constants. The last solution is based on maintenance of the vector of possible states by each agent, masking the actual state of the swarm (defined to be the one with a majority of copies) by redundant states (with fewer copies). In general, two states maintained by an agent may yield the same next state when a certain input

is received, thus redundancy of states may be eliminated over time. Randomization is used in [6] in order to cope with such a convergence, randomly choosing a state for the extra copies in the vector when two or more states become equal. In contrast, in this work we show that it is possible to solve the problem of convergence to the same state in a deterministic way using Krohn-Rhodes decomposition. Furthermore, the scheme in this work is information-theoretically secure.

As mentioned above, the authors recently presented schemes that support the same type of *perennial* private computation considered here by implementing a universal Turing machine privately, with one round of communication per transition [4]. In this work we show how to avoid communication completely during the course of the actual execution, at the expense of computing a smaller class of functions.

The type of private computation we consider is also related to the problem of (information-theoretic, or unconditional) secure multi-party computation (MPC) [1, 3]. We perform a detailed comparison below.

1.4 Unbounded-input private computation vis-à-vis MPC

Recall that in secure multi-party computation, n parties ("players"), some of which might be corrupted, are to compute an n-ary (public) function on their inputs, in such a way that no information is revealed about them beyond what is revealed by the function's output. At a high level, we similarly aim in our context to ensure the correctness and privacy of the distributed computation. However, as we now argue, our setting is significantly different from that of MPC, and MPC solutions cannot be directly applied here.

Firstly, MPC aims to solve a different problem, that of protecting the players' individual inputs from *Adv*, who can corrupt some of them, learn their input and observe the communication they receive. In contrast, in our problem the inputs are common to all the players (but not *a priori* known to *Adv*, or revealed in case of corruption), and the goal is to protect the state of, as well as the inputs to the computation. (Therefore, we cannot in particular treat the common input as public information, and the shares received from the dealer as MPC input.)

Of course, an adequate representation (circuit-based, for example) of the MPC computation would be able to evaluate $\mathcal{A}$, with respect to a subset of corrupted players, and at least for the basic MPC setting, where there is a single (tuple of secret) input(s) out of which an output (tuple) is produced. But then comes our main feature, of multiple, possibly unbounded number of input symbols. This is reminiscent of secure *reactive* systems (e.g., [15]), where the computation is not limited to "one shot" as above, but instead processes inputs "in blocks" throughout several rounds of interaction. However, because all MPC solutions (and definitions) are explicitly tied to the length of the input, being able to handle unbounded number of inputs without communication does not seem immediate. This is what our Krohn-Rhodes-based approach achieves, at the expense of solving a narrower problem.

Looking at the relationship with MPC from another perspective, we note that it is the *combination* of our requirements of non-interactivity during the input-processing phase, information-theoretic security, and computation on inputs of unbounded length, that precludes the use of known MPC techniques. That is, with any of the three requirements removed, known techniques would allow stronger results to be achieved.

Indeed, we have discussed above the possibility of solutions in the setting where the inputs are bounded. Alternatively, if we only required computational secrecy, then the players could use fully homomorphic encryption [9] to maintain under encryption the current state of the computation on unbounded inputs (and carry shares of the scheme's private key). Further, if an *a priori* bound on the input length existed, players could simply encrypt their inputs with any public-key encryption scheme, again keeping shares of the decryption key. Finally, allowing interaction during the input processing phase can effectively bring us to the bounded-input setting, since interaction—and thus share updates using MPC—could occur after a certain fixed number of inputs has been processed.

1.5 Organization of the paper

Section 2 presents the necessary background for this paper. Section 3 defines our notion of secure computation in a swarm and Section 4 describes our construction in detail. For the purpose of readability, proofs are presented in an appendix.

2 Preliminaries, notation and background

In this section we introduce the notation used throughout the paper, and present the necessary background material and tools—secret sharing schemes, finite state automata, and Krohn-Rhodes theory. Let $P_1, \ldots, P_n$ be the n agents that distributively will execute $\mathcal{A}$.

2.1 Secret sharing

We start with an overview of our basic tool, *secret sharing* [17], where essentially, a secret piece of information is "split" into shares by a distinguished player called *the dealer*, in such a way that up to a threshold $t < n$ of the players pulling together their shares are not able to learn anything about it, while $t+1$ are able to reconstruct the secret. In fact, we consider general secret sharing for any *monotone access structure* [11], a generalization of threshold secret sharing. More formally, an *access structure* $\mathcal{U}$ is simply a set of subsets of $P_1, \ldots, P_n$, that is, $\mathcal{U} \subseteq 2^{\{P_1,\ldots,P_n\}}$. We say that $\mathcal{U}$ is *monotone* if for every $\mathcal{I} \in \mathcal{U}$, we have that $\mathcal{I}' \in \mathcal{U}$ for every $\mathcal{I}'$ such that $\mathcal{I} \subseteq \mathcal{I}'$.

Definition 2.1 *We say that a secret-sharing scheme S has an* access structure $\mathcal{U}$*, if any set of shares held by players of any set $\mathcal{I} \in \mathcal{U}$ allows the reconstruction of the secret, while shares held by players $\mathcal{I}' \notin \mathcal{U}$ yields no information on the secret. Additionally, we say that S has a monotone access structure, if $\mathcal{U}$ is monotone.*

For simplicity, sometimes in our discussion we concentrate on (t, n) threshold secret sharing, where $\mathcal{U}$ is the set of all subsets of the n players of size greater than t. Further, the presentation above is only concerned with the privacy of the secret. Correctness in our scenario can also be guaranteed against actively disruptive behavior by at most t parties by employing so-called *unverified* secret sharing schemes [5, 16] ("unverified" relates to the fact that the dealer is honest), and adjusting the threshold (resp., access structure) to $t < \frac{n}{3}$.

2.2 Some automata theory notions

A *finite-state automaton* (FSA) $\mathcal{A}$ has a finite set of states ST, a finite set of input symbols Γ, and a transition function $\mu : ST \times \Gamma \longrightarrow ST$. We do not assume an initial state or a terminal state; the automaton may begin its execution from any state and does not necessarily stop. $\mathcal{A}' = (ST', \Gamma\, \mu')$ is a *sub-automaton* of $\mathcal{A}$ if $ST' \subseteq ST$, $\Gamma' \subseteq \Gamma$ and μ' is the reduction of μ to $ST' \times \Gamma'$ and $\mu'(s', \gamma') \in ST'$, for any $s' \in ST'$ and any $\gamma' \in \Gamma'$. For every input symbol $\gamma \in \Gamma$ and every $s \in ST$, we denote by $\mu_\gamma : ST \longrightarrow ST$ the function $\mu_\gamma(s) = \mu(s, \gamma)$. Further, we denote the state of $\mathcal{A}$ when executed with initial state s_{init} and input $\gamma_1 \ldots, \gamma_k$ by $\mathcal{A}(s_{init}, \gamma_1 \ldots, \gamma_k)$.

Definition 2.2 *A* permutation automaton *is a finite automaton such that for every $\gamma \in \Gamma$, the function μ_γ is a permutation on ST.*

Definition 2.3 *A* flip-flop automaton *is a finite automaton with two states $ST = \{s_0, s_1\}$ and three inputs $\Gamma = \{\gamma_0, \gamma_1, \gamma_2\}$, such that μ_{γ_0} is the identity function, μ_{γ_1} is defined by $\mu_{\gamma_1}(s_0) = \mu_{\gamma_1}(s_1) = s_0$, and μ_{γ_2} is defined by $\mu_{\gamma_2}(s_0) = \mu_{\gamma_1}(s_1) = s_1$.*

The left side of Figure 1 is an example of a *permutation automaton*. The right side of Figure 1 shows a flip-flop automaton.

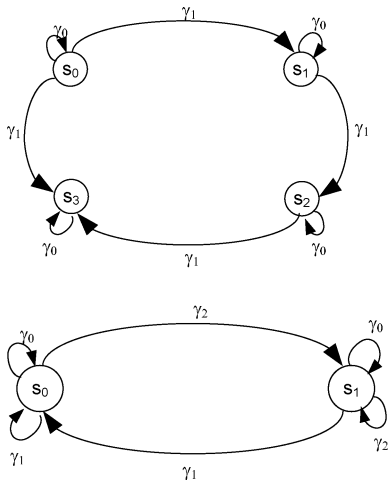

Figure 1: Examples of permutation and flip-flop automata.

A *cascade product* of automata is a sequence of automata such that the input to the i-th automaton is a function of a global input and of the states of automata $1, 2, \ldots, i-1$. As an example consider the automaton in Figure 2, that starting at state s_0 reaches state s_a if and only if four γ_1 symbols appear in its input. This Four-γ_1 checker can be emulated by a sequence of two automata: the permutation automaton in Figure 1 and the flip flop of Figure 1.

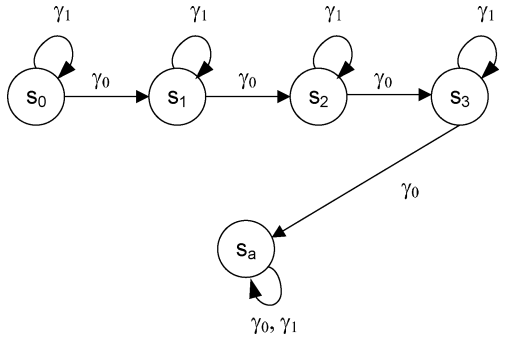

Figure 2: Four γ_1 checker.

Let the permutation automaton and the flip flop begin execution from s_0. The input of the permutation automaton is the same input as that of the four γ_1 checker. The input of the flip-flop is determined by a function Ψ that maps a state of the permutation automaton and an input symbol of the four γ_1 checker to an input symbol of the flip-flop. $\Psi(s_3, \gamma_0) = \gamma_2$, and for any other input pair $\langle s, \gamma \rangle$, we define $\Psi(s, \gamma) = \gamma_0$. This cascade product emulates the four γ_1 checker because the flip flop is in state s_1 if, and only if, the four γ_1 checker is in state s_a while the flip flop is in state s_0; thus the permutation automaton is in the same state as the four γ_1 checker.

2.3 Krohn-Rhodes theories

At a high level, the hierarchical algebraic decomposition of finite state automata, known as Krohn-Rhodes theory [12, 13], shows how to emulate a finite automaton by a combination (*product*—see below) of permutation and flip-flop automata. We now present the background on Krohn-Rhodes theory that is necessary for the current work. Our presentation follows the interpretation given in [7], which greatly expands on the short summary we present, and by Margolis in [14].

Definition 2.4 *Let $\mathcal{A} = (ST, \Gamma, \mu)$ and $\mathcal{A}' = (ST', \Gamma', \mu')$ be two automata. A pair $\Psi = (\Psi_1, \Psi_2)$ of surjective mappings $\Psi_1 : ST \longrightarrow ST'$ and $\Psi_2 : \Gamma \longrightarrow \Gamma'$ is a* homomorphism *of $\mathcal{A}$ onto $\mathcal{A}'$ if for every $s \in ST, \gamma \in \Gamma$ we have $\Psi_1(\mu(s, \gamma)) = \mu'(\Psi_1(s), \Psi_2(\gamma))$.*

If $\mathcal{A}$ has a sub-automaton which can be mapped homomorphically by Ψ onto $\mathcal{A}'$, then we say that $\mathcal{A}$ *homomorphically represents* $\mathcal{A}'$. An important property of such homomorphism is that $\mathcal{A}$ can be executed instead of $\mathcal{A}'$ in the following sense. Suppose $\mathcal{A}'$ begins execution at state s', receives a stream of input symbols $\gamma'_1, \ldots, \gamma'_k$, and terminates in state t'. Then, if $\mathcal{A}$ begins execution in state s, such that $\Psi_1(s) = s'$ and receives a stream of input symbols $\gamma_1, \ldots, \gamma_k$ such that $\gamma_i = \Psi_2(\gamma_i)$ for every i then $\mathcal{A}$ terminates in state t such that $\Psi_1(t) = t'$.

There are several different ways to arrange a sequence of automata, $\mathcal{A}_1 = (ST_1, \Gamma_1, \mu_1)$, ..., $\mathcal{A}_m = (ST_m, \Gamma_m, \mu_m)$, to define a single *product* automaton. We are interested in the following type of product.

Definition 2.5 *A finite automaton $\mathcal{A} = (ST, \Gamma, \mu)$ is called a* cascade product *of $\mathcal{A}_1 = (ST_1, \Gamma_1, \mu_1)$, ..., $\mathcal{A}_m = (ST_m, \Gamma_m, \mu_m)$, denotedby $\mathcal{A}_1$, ..., $\mathcal{A}_m(\Gamma, \varphi_1, \ldots, \varphi_m)$, if $ST = ST_1 \times \ldots \times ST_m$, there exist functions $\varphi_1, \ldots, \varphi_m$ such that $\varphi_i : ST_1 \times \ldots \times ST_{i-1} \times \Gamma \longrightarrow \Gamma_i$, for every $i = 1, \ldots, m$, and μ is defined by $\mu((s_1, \ldots, s_m), \gamma) = (\mu_1(s_1, \varphi_1(\gamma)), \ldots, \mu_m(s_m, \varphi_m(s_1, \ldots, s_{m-1}, \gamma))$.*

Thus, in a cascade product, the input to the ith component automaton ("child" automaton) is a function of the global input (γ) and the $i-1$ states of the previous $i-1$ cascade components ("parent" automata).

Let $\mathcal{A} = (ST, \Gamma, \mu)$ be an automaton and denote by μ_w a function on the states defined by $\mu_w(s) = \mu(s, w)$ for every $w \in \Gamma^*$ (that is, words defined over Γ). The *characteristic semigroup* of $\mathcal{A}$ is $S(\mathcal{A}) \stackrel{\triangle}{=} \{\mu_w : w \in \Gamma^+\}$ with composition of functions as the semigroup's binary operator. $S(\mathcal{A})$ together with μ_λ (where λ is the empty word) forms the characteristic monoid. If this monoid is a group, then $\mathcal{A}$ is a permutation automaton. $S(\mathcal{A})$ is a *transformation semi-group*, i.e., it includes functions of ST into itself. We denote this transformation semi-group by $(ST, S(\mathcal{A}))$.

Definition 2.6 $(ST, S(\mathcal{A}))$ divides $(ST', S(\mathcal{A}))'$ *if for some subset $Y \subseteq ST'$, and sub-semigroup T of $S(\mathcal{A}))'$ that maps Y into itself, there exists an onto function $\theta_2 : Y \longrightarrow ST$ and a surjective semigroup homomorphism $\theta_1 : T \longrightarrow S(\mathcal{A}))$ satisfying $\theta_2(t(y)) = \theta_1(t)(\theta_2(y))$ for all $y \in Y, t \in T$.*

Theorem 2.7 (Krohn-Rhodes) *A finite automaton $\mathcal{A}$ can be homomorphically represented by a cascade product of components from $\{\mathcal{A}_F, \mathcal{A}_{G_1}, \ldots, \mathcal{A}_{G_\ell}\}$, where $\mathcal{A}_F$ is a flip-flop automaton and $\mathcal{A}_{G_1}, \ldots, \mathcal{A}_{G_\ell}$ are permutation automata. Furthermore, if $S(\mathcal{A})$ is the characteristic semi-group of $\mathcal{A}$ and $G_1, \ldots, G_\ell$*

are the characteristic groups of $\mathcal{A}_{G_1}, \ldots, \mathcal{A}_{G_\ell}$*, then* $G_1, \ldots, G_\ell$ *can be chosen as all the simple groups that divide* $S(\mathcal{A})$*.*

To be concrete, we will be interested in the most efficient method of constructing a Krohn-Rhodes-type cascade product, namely, Eilenberg's *holonomy decomposition* [8, 19]. In holonomy decomposition, an automaton is decomposed into component automata that have two kinds of transitions: permutations and resets; a reset transition is simply the constant function from all the states to one single state. Specifically, each component $\overline{\mathcal{H}}$ in a holonomy decomposition includes a permutation automaton $\mathcal{H}$ and all resets on $\mathcal{H}$.

Theorem 2.8 (Holonomy decomposition [8]) *A finite automaton* $\mathcal{A}$ *with* m *states can be homomorphically represented by a cascade product of components* $\overline{\mathcal{H}}_1, \ldots, \overline{\mathcal{H}}_k$*, such that* $k \leq m$ *and the number of states in* $\mathcal{H}_i$*,* $i = 1, \ldots, m$*, is at most* $m - i + 1$*.*

In our scheme, we will require a decomposition into permutation automata and flip-flops (which have resets on two states). Transforming an i-state component $\overline{\mathcal{H}}$ into a cascade product of permutation automata and flip-flops is easy by using $\mathcal{H}$ and $\log_2 i$ flip-flops to represent all the resets on $\mathcal{H}$.

3 Private distributed computation on global inputs

In this section we define the exact notion of private distributed computation in our context. Recall that our goal is for agents $P_1, \ldots, P_n$ to jointly compute a state of a given finite automaton $\mathcal{A}$ on the (*a priori* unbounded) global input stream X received by each agent, in such a way no t of them reveal any information about it. The agents' computation is divided into three phases: sharing phase, online (computation) phase, and reconstruction. In the sharing phase, the dealer initializes the agents based on his random tape, the automaton $\mathcal{A}$, and its initial state s_{init}. In the online phase, the agents process the inputs and update their state accordingly; in this phase, no communication takes place, and the agents may be corrupted by *Adv*. Finally, in the reconstruction phase, the agents collaborate to reconstruct the state of $\mathcal{A}$. We stress that it is assumed that *Adv* is not able to interfere with or observe the sharing and reconstruction phases.

As noted above, in the online phase, up to a threshold t of the agents[3] may be corrupted by the adversary *Adv*, who then learns their internal states. We require that the agents carry out the computation without leaking any information, such as initial and current states and inputs seen. For simplicity, we consider *semi-honest* (also called "honest-but-curious") adversaries; that is, each participant (even when corrupted) continues to execute the protocol as required. Note, however, that since the online phase does not involve any communication, an actively malicious behavior would be of no consequence and only amount to inspection of the states of corrupted players. During the reconstruction, disruptive behavior and "cheating" can be overcome simply by using secret sharing with stronger reconstruction (lower threshold) properties. Informally, we say that the distributed computation performed by the agents as described above is *secure* if it is correct, and the combined state of up to the threshold of corrupted parties gives no information about the state of the computation or on the history of inputs.

We define two notions of security in this work. Security in a weaker setting, which we call the *simultaneous corruption* model, relates to the adversary who is only allowed to corrupt up to t parties simultaneously. Although weaker than the next, this is perhaps a more intuitive model, which is also practical and interesting in its own right. We then consider the stronger *progressive corruption* model, where the adversary is allowed to corrupt players as the execution of the protocol proceeds[4]. Our protocols are secure in the stronger progressive corruption model.

As mentioned before, we allow the adversary to have unbounded computational power, and are thus interested in protocols that are information-theoretically secure. Our definitions follow the standard paradigm in cryptography of indistinguishability/simulatability of views of the adversary.

Let $\mathcal{A} = (ST, \Gamma, \mu)$ be an FSA with initial state $s \in ST$. Let $X = (\gamma_1, \gamma_2, \ldots) \in \Gamma^*$ be the (global) input given to all agents. Let $\mathcal{I} = \{P_{i_1}, \ldots, P_{i_\ell}\}$, $\ell \leq t$, be a subset of agents. We denote by $\mathrm{View}^{\Pi}_{\mathcal{I}}(X, s)$ the probability distribution of the view of *Adv*, which is

[3] Our protocols and definitions can be readily extended to general access structures.

[4] We note that our two allowed corruption models resemble the secure multi-party computation (MPC) notions of *static* and *adaptive* security, respectively [2]. However, since in our setting the adversary is not allowed to corrupt agents at the onset of the computation, as well as to continue to monitor their state after corruption, as is the case in MPC, we use different names to avoid confusion.

the aggregated memory contents of all parties in $\mathcal{I}$ as they execute protocol Π on input X starting on $\mathcal{A}$'s initial state s.

Definition 3.1 (Correctness) *We say that protocol* Π evaluates *automaton* $\mathcal{A}$ *with respect to threshold* t *if for any natural number* k *and any input stream* $\gamma_1 \ldots, \gamma_k$ *as above, (only) the members of any subset* $\mathcal{S} \subset \{P_1, ..., P_n\}$, $|\mathcal{S}| > t$, *correctly output* $\mathcal{A}(s_{init}, \gamma_1 \ldots, \gamma_k)$ *in its reconstruction phase.*

Definition 3.2 (Privacy in the SCM) *We say that* Π *is* private in the Simultaneous Corruption Model (SCM) *if for every two states* $s_1, s_2 \in ST$, *input streams* $X_1, X_2 \in \Gamma^*$, *and sets of agents* $\mathcal{I}_1, \mathcal{I}_2$, $|\mathcal{I}_1| = |\mathcal{I}_2| \leq t$, $\text{View}^{\Pi}_{\mathcal{I}_1}(X_1, s_1) = \text{View}^{\Pi}_{\mathcal{I}_2}(X_2, s_2)$.

We now give some intuition behind Definition 3.2. The simultaneous corruption model addresses the case where *Adv* takes a "snapshot" of the states of corrupted players at an arbitrary point during the online phase. Our requirement of identical distribution of views means that the combined memories of the corrupted players are independent of the so-far processed inputs, initial states, and even of the IDs of the corrupted players (note that they are not included in the view). As an instructive example, let us verify that the definition guarantees that the current state of $\mathcal{A}$ is hidden from *Adv*. Indeed, suppose otherwise, and that *Adv* can recover some information about the state of $\mathcal{A}$ if a certain input sequence is executed. Then, *Adv* could distinguish this view from, for example, the view resulting from the execution on an empty input on a random initial state, which is prohibited by the definition.

Next, we introduce a stronger notion of security, which allows *Adv* to corrupt the agents as the execution progresses. First, we define a *corruption timeline* as the vector $\rho = (\langle n_1, P_{i_1}\rangle, \langle n_2, P_{i_2}\rangle, ..., \langle n_\ell, P_{i_\ell}\rangle)$, where $n_j \in \mathbb{N}$, $1 \leq j \leq \ell$, is the number of symbols that are received before P_{i_j} is corrupted, and $\ell \leq t$. For clarity, we state that $P_{i_j} \neq P_{i_k}$ for $i \neq k$, i.e., a player cannot be corrupted twice. We denote by $\text{View}^{\Pi}_{\rho}(X, s)$ the probability distribution of the aggregated internal states of corrupted agents at the time of corruption, when executed on input $X \in \Gamma^*$ and initial state s.

Definition 3.3 (Privacy in the PCM) *We say that* Π *is* private in the Progressive Corruption Model (PCM) *if for every two states* $s_1, s_2 \in ST$, *input streams* $X_1, X_2 \in \Gamma^*$, *and corruption timelines* ρ_1, ρ_2, $|\rho_1| = |\rho_2|$, $\text{View}^{\Pi}_{\rho_1}(X_1, s_1) = \text{View}^{\Pi}_{\rho_2}(X_2, s_2)$.

This definition follows the spirit of Definition 3.2, with the notable addition of the power of *Adv* to corrupt agents at different stages of execution of the online phase, perhaps more natural in some scenarios than taking a simultaneous "snapshot" of the agents' internal states. Yet, by not allowing the adversary in this corruption model to see the subsequent inputs of a corrupted agent, we are able to impose a stronger security requirement, namely, the indistinguishability of the views generated by two different input streams.

Remark 3.4 *In our definitions, we require that the views of adversary based on any two executions be distributed identically. Clearly, this implies that it is possible to simulate the views without knowing the dealer's shares, inputs, and even the length of the input stream. The simulation is simply performed by executing the protocol on any (e.g., randomly chosen) input. Further, the complexity of the simulator is exactly equal to that of an honest player executing the potocol.*

4 Privately computing an automaton state

In this section we present our automata-based construction, which allows us to achieve the notion of distributed private computation formulated in Section 3; the construction is based on the Krohn-Rhodes decomposition. We first present a high-level intuition of how it works.

4.1 The construction at a high level

Assume we are given a cascade product of permutation and flip-flop automata. We show how to secret-share the composed automaton and its current state, and how to update the shares as a result of the agents receiving (an unbounded number of) input symbols.

At any time, a quorum of secret-sharing agents can reconstruct the state of the automaton. Recall that the automaton itself is public, and need not be protected.

An (independent) permutation automaton $\mathcal{A}$ is shared simply by secret-sharing among the agents a 1 for the active state, and a 0 for each other state. Thus, for an automaton with m states, each participant will have m shares, one share associated with

each state of $\mathcal{A}$. Upon input $\gamma \in \Gamma$, the player will simply reassign each share according to γ. That is, for each state s_i, the share associated with s_i will be assigned to state $\mu(\gamma, s_i)$. It is easy to see, from the properties of secret sharing, that such sharing and input processing maintains the privacy of current states and the correctness of state reconstruction.

An (independent) flip-flop automaton $\mathcal{A}$ is shared and processed as follows. As $\mathcal{A}$ consists of only two states, we simply create a "mirror" flip-flop $\mathcal{A}'$, which, on any sequence of inputs, always remains in the state opposite to that of $\mathcal{A}$. Then, the dealer sends each player the two automata in a random order, and a share of a pointer to the "correct" one. Upon input $\gamma \in \Gamma$, the agents simply apply the input to each of the two automata they have. Clearly, the secrecy of the current state is preserved; further, correctness of reconstruction is assured since the quorum of agents can reconstruct the pointer to the correct automaton, and obtain the current state.

However, recall that we are interested in the cascade product representation, where the the input for each component automaton depends on the states of its "parent" automata in the cascade (see Definition 2.5 and the following paragraph). Clearly, the agents must not know any of the automaton's states. Our solution is to maintain many instances of each child automaton (one for each state configuration of parent automata). Then we can apply the transition function based on the state associated with each particular instance. Of course, as inputs are processed and the parent state configuration changes, we need to reassign these configurations to the child instances.

One natural way to do the above is by representing the automaton composition as a tree, rooted with the upper level automaton, $\mathcal{A}_1$. In the tree, each parent permutation automaton $\mathcal{A}_i$ has m_i children (one for each of its states). A parent flip-flop automaton $\mathcal{A}_i$ has two child automata. Each of the children is labeled (via the tree edge) with the state of $\mathcal{A}_i$ it is associated with. Thus, a labeling of the path to the root represents the entire parent configuration, and allows the players to apply the transition function to each (share of the) automaton in the tree.

Finally, as the input is processed, we maintain the correct associations between child instances and parent states by reassigning the edge labeling according to the parent state transition. That is, for each state transition $s_i \to \mu(\gamma, s_i)$ in the parent automaton, each edge s_i is relabeled with $\mu(\gamma, s_i)$. This ensures that the child, which has a state that depends on the state of its immediate parent, continues to properly "follow" the parent's state. Because the parent's labels are also adjusted, each child automaton consistently "follows" its parent configuration.

State reconstruction for this scheme is also natural. Given the quorum, we start by reconstructing the parent state, and then proceed with the reconstruction down the appropriate path.

4.2 The construction in detail

Initialization phase. The dealer begins this phase by computing a cascade product of permutation automata and flip-flops, $\mathcal{A}_1, \ldots, \mathcal{A}_m(\Gamma, \varphi_1, \ldots, \varphi_m)$ that homomorphically represents $\mathcal{A}$. Krohn-Rhodes theory (cf. Section 2) states that such a cascade exists, and holonomy decomposition is a way to construct it.

The dealer constructs a rooted tree, $\mathcal{T}$, based on this cascade. The tree consists of $m+1$ levels, level 1 for the root and level $m+1$ for the leaves. Each node in levels $1, 2, \ldots, m$ contains an automaton, while all the nodes in level $m+1$ are empty. If $\mathcal{A}_i$ is a permutation automaton, then all level i nodes contain $\mathcal{A}_i$. If $\mathcal{A}_i$ is a flip-flop automaton, then every node at level i contains two flip-flops.

In each node that holds two flip-flops, there is one flip-flop such that the initial state is s_0, γ_0 is the identity function, γ_1 resets to s_0 and γ_2 resets to s_1. In the other flip-flop, the initial state is s_1, and the reset edges are switched: γ_1 resets to s_1 and γ_2 resets to s_0. This arrangement ensures that if both flip-flops receive the same input they will always be in opposite states.

A node in level i, $1 \leq i \leq m$, has several children in level $i+1$. If $\mathcal{A}_i = (ST_i, \Gamma_i, \mu_i)$ is a permutation automaton then the node has $|ST_i|$ children, a child for each state of $\mathcal{A}_i$. If $\mathcal{A}_i$ is a flip-flop, then the node has two children, one for each of the two flip-flops in the node. Each edge between parent and child has a value that marks, or labels it. If $\mathcal{A}_i$ is a permutation automaton, then the label is the state with which the edge is associated. This label may change after each transition of the automaton. If $\mathcal{A}_i$ is a flip-flop, then the label indicates which of the two flip-flops is associated with this edge. This label does not change during execution of the automaton.

The dealer distributes to every player the func-

tions $\varphi_1, \ldots, \varphi_m$ to compute transitions based on input symbols for each component of the cascade.

Since $\mathcal{A}_1$, ..., $\mathcal{A}_m(\Gamma, \varphi_1, \ldots, \varphi_m)$ homomorphically represents $\mathcal{A}$, there exists a homomorphism $\Psi = (\Psi_1, \Psi_2)$ that maps an m-tuple of states in $\mathcal{A}_1$, ..., $\mathcal{A}_m$ to a state in $\mathcal{A}$. Let the state set of $\mathcal{A}_i$ be ST_i for every $i = 1, \ldots, m$. Then, the dealer secretly shares Ψ among the n players by sharing the value of $\Psi_1(s_1, \ldots, s_m)$ for every tuple of states $(s_1, \ldots, s_m)$, $s_i \in ST_i$.

The dealer completes the initialization stage by sharing secrets. Let s be the secret initial state of $\mathcal{A}$. Since the cascade $\mathcal{A}_1$, ..., $\mathcal{A}_m(\Gamma, \varphi_1, \ldots, \varphi_m)$ homomorphically represents $\mathcal{A}$ there is a sequence of states $(s_1, \ldots, s_m) \in \mathcal{A}_1 \times \ldots \mathcal{A}_m$, such that the homomorphism maps $(s_1, \ldots, s_m)$ to s. Thus, the dealer regards s_i as the initial state of $\mathcal{A}_i$. If $\mathcal{A}_i$ is a permutation automaton then the dealer shares value 1 for s_i and 0 for any other state in $\mathcal{A}_i$. A player denotes its share of the value for a state s by Share(s) and uses it to label the edge marked by s. If $\mathcal{A}_i$ is a flip-flop then the dealer shares value 1 for the correct flip-flop and shares 0 for the opposite flip-flop. A player denotes its share of the value for a flip-flop FF by Share(FF) and uses it to label the edge marked by FF. Each player attaches all the shares associated with $\mathcal{A}_i$ to every node in level i of $\mathcal{T}$.

Online phase. We now describe the online computation that each player performs. The basic idea is to view each path from root to leaf in $\mathcal{T}$ as a cascade product of automata. Given an input symbol for $\mathcal{T}$, each such cascade executes a step. A step of a permutation automaton is represented by moving the marks on each edge according to the transition of automaton states. A step of a flip-flop is represented by moving the state of each of the two flip-flops in a node. Algorithm 1 shows the processing performed by an agent upon receiving an input symbol γ.

Reconstruction phase. After the online phase, executing Algorithm 1 on a (possibly unbounded) number of input symbols, a subset of agents in the access structure run Algorithm 2 to reconstruct the current state of $\mathcal{A}$ from $\mathcal{T}$ and their shares.

4.3 Analysis of the construction

We first show that our construction satisfies Definition 3.1, i.e., using the initialization procedure of Section 4.2 together with running Algorithms 1 and 2 ensures that the players $P_1, \ldots, P_n$ correctly compute a state of $\mathcal{T}$. For simplicity, we state and prove the result for passive ("honest-but-curious") corruptions; in the case of active corruptions, "all agents" would be replaced by "all uncorrupted agents," and setting the threshold of allowed corruptions accordingly. Next, in Proposition 4.2 we show the privacy of the construction; we directly address the stronger notion of security.

Proposition 4.1 *Let $\mathcal{A} = (ST, \Gamma, \mu)$ be the automaton held by the dealer, and s_{init} an initial state. Fix a natural number k, and a sequence of k input symbols $(\gamma^1, \ldots, \gamma^k)$, $\gamma^j \in \Gamma$, $j = 1, \ldots, k$. Assume that executing $\mathcal{A}$ on $(\gamma^1, \ldots, \gamma^k)$ from state s_{init} terminates in state s_{term}. Then performing the initialization procedure described in Section 4.2, and all agents $P_1, \ldots, P_n$ executing Algorithm 1 on every γ^j and Algorithm 2 after γ^k is processed, returns the state s_{term}.*

Algorithm 1 Online Phase (γ)

1: Go over every node in $\mathcal{T}$ beginning at the root in order of Depth First Search.
2: Let N be a node at level i of the tree and let N_1, ..., N_{i-1} be the sequence of nodes from the root to N.
3: **for** $j = 1$ to $i - 1$ **do**
4: **if** $\mathcal{A}_j$ is a permutation automaton **then**
5: Set s_j to be the state marking the edge from N_j to N_{j+1}.
6: **else**
7: Set s_j to be the current state of the flip-flop marking the edge from N_j to N_{j+1}.
8: Compute $\gamma_N \leftarrow \varphi_i(s_1, \ldots, s_{i-1}, \Psi_2^{-1}(\gamma))$ and store γ_N in N.
9: **for all** nodes N **do**
10: **if** N holds a permutation automaton, say, $\mathcal{A}_i = (ST_i, \Gamma_i, \mu_i)$ **then**
11: **for all** edges from N to its children **do**
12: Change the state marking this edge from s to $\mu_i(s, \gamma_N)$, but do not change the secret share.
13: **else**
14: Change the current state of the first flip-flop from s to $\mu_i(s, \gamma_N)$.
15: Change the current state of the second flip-flop from s' to $\mu_i(s', \gamma_N)$.

Proposition 4.2 *The construction of Section 4.2—initialization procedure, and Algorithms 1 and 2—guarantees privacy in the progressive corruption model, according to Definition 3.3.*

Algorithm 2 Reconstruction of current state

1: Set N to be the root of $\mathcal{T}$.
2: **for** $i = 1$ to m **do**
3: **if** N holds a permutation automaton $\mathcal{A}_i = (ST_i, \Gamma_i, \mu_i)$ **then**
4: **for all** edges from N to its children **do**
5: Let the subset of players reconstruct the secrets marking those edges.
6: Set a variable $s_i \leftarrow s$, where s is the only edge marked with the secret value 1.
7: **else**
8: Let the subset of players reconstruct the secrets marking the two edges from N to its children.
9: Set a variable s_i to be the current state of FF, where FF is the flip-flop associated with the only edge marked with a secret value 1.
10: The subset of players returns $\Psi_1(s_1, \ldots, s_m)$ as the current state of $\mathcal{A}$.

Regarding the complexity of our construction, we consider two measures: the space complexity, which is dominated by the number of states in the tree $\mathcal{T}$, and the computational complexity, given by the number of steps we perform in $\mathcal{T}$ for each transition of $\mathcal{A}$; this second measure is identical to the number of separate automata in $\mathcal{T}$.

Given a cascade product $\mathcal{A}_1, \ldots, \mathcal{A}_m(\Gamma, \varphi_1, \ldots, \varphi_m)$, our scheme constructs a tree of depth m such that the number of children for each node of depth i is $|ST_i|$. Thus, the number of states in the leaves of our tree is exactly the number of states in the cascade product, that is, $\prod_{i=1}^{m} |ST_i|$. Since each inner node in the tree has at least two children, the total number of states is at most twice the number of leaves. The number of different automata in the leaves is $\prod_{i=1}^{m-1} |ST_i|$.

The remaining question is relating the size of a cascade product to the original $\mathcal{A}$. As stated previously, holonomy decomposition is the best general construction method known. If the number of states in $\mathcal{A}$ is $|ST| = m$, then the total number of states resulting from the decomposition is the product of the number of states of each component, and is thus $m!$. The total number of states in $\mathcal{T}$ is therefore at most $2m!$, and the total number of automata is at most $2(m-1)!$.

In some cases, holonomy decomposition gives exponentially more states than the optimum. As an example, consider an automaton $\mathcal{A}$ that enters a sink state s if and only if it is given a sequence of n consecutive symbols γ. Decomposing that automaton is possible with one permutation automaton and one flip-flop. However, holonomy decomposition would decompose it into m flip-flops. In the first case, $\mathcal{T}$ is of size $O(m)$, while in the latter it is of size $O(2^m)$. Not to despair, in the next section we present a natural automata family admitting a small Krohn-Rhodes decomposition.

4.4 KR decomposition example

Consider a decision tree of depth d and fan-out at most k. Each edge in the tree is marked with a character in a finite alphabet that denotes some external input. The edges connecting a node N to its children are all marked by different characters. In such a tree, one reaches a decision by beginning at the root and traversing a path to a leaf which stores some decision. At each node the choice of which child to choose next depends on an input which defines the next edge, and thus the "right" child. Such trees are commonly used in various branches of artificial intelligence, natural language processing and other disciplines.

Consider a generalization of such a decision tree, given by adding to each node N a loop edge back to N. In every internal node, such a loop determines the action to take when the input is not one of the characters marking an edge to a child of N. In every leaf, this loop is marked by every character in the input alphabet. This construction yields an automaton that emulates a decision tree that can wait as long as required for the right input to reach decisions. Figure 3 shows an example of such a generalized tree, that has depth $d = 3$, fan-out $k = 2$, accepts as input the set $\{1, 2, 3, 4\}$ and is currently in the right node of the second level.

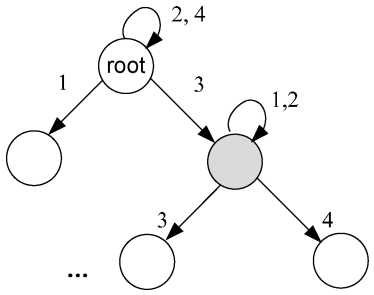

Figure 3: Generalized decision tree.

We next prove that such a generalized decision tree has a Krohn-Rhodes decomposition with $(d-1)(1 + \log k)$ flip-flops. As a first step we transform the generalized decision tree into a cascade product of $d-1$ automata, where each automaton represents one in-

ternal level of the decision tree. Each such automaton has $k+1$ nodes. The first k nodes represent the (at most) k children of a node at the current level, while the last node, number $k+1$ represents the current level.

Each node in the generalized decision tree is mapped to a product of $d-1$ nodes in the cascade product. A node in the decision tree that is reached from the root by a path $s_1, \ldots, s_j$, where $j < d$ is represented by the tuple $(s_1, \ldots, s_j, k+1, \ldots, k+1)$ in the cascade product.

The automaton representing the root accepts every input that the decision tree accepts. The first k nodes of this automaton loop to the same node for every input (in other words, once the state of the decision tree moves from the root to one of its children, it will never move to a different child). Node $k+1$ has an edge to node i, $i = 1, \ldots, k$ marked with any input that sends the root to the i-th child in the decision tree. For any other input node $k+1$ loops back into itself.

In the automaton representing level j, $1 < j < d$, there are $k+1$ possible inputs $\gamma_1, \ldots, \gamma_{k+1}$. The first k nodes loop to the same node for every input. Node $k+1$ has an edge marked γ_i to node i for every $i = 1, \ldots, k$ and loops back into node $k+1$ with input γ_{k+1}. Thus input γ_i for $i = 1, \ldots, k$ means that the current state of the decision tree moves to the i-th child, while input γ_{k+1} means that the state does not change.

Computing an input for an automaton in the j-th level of the cascade product is done in the natural way: given that the first j_1 automata are in states $s_1, \ldots, s_{j-1}$ and that the input to the cascade product is δ we emulate the decision tree when its state is reached by a path $(s_1, \ldots, s_{j-1}$ from the root and its input is δ. If the decision tree moves to child i, the input to the j-th automaton is γ_i and if the state of the decision tree does not change on input δ then the input to the j-th automaton is γ_{k+1}.

Each of the automata in the cascade product have $k+1$ nodes and can be represented by a cascade product of $1+\log k$ flip-flops. The first flip-flop in the product determines whether the automaton is in state $k+1$ or in one of the other k states. This flip-flop has the identity transition and a reset to the state representing the k children. The other $\log k$ flip-flops represent which child is chosen.

Figure 4 shows the flip-flops and the current states of a decomposition for the decision tree in Figure 3. For each of the first $d-1=2$ levels of the tree there are $1+\log k = 2$ flip-flops, thus four automata in this decomposition. The state of the first automaton in each level shows whether the current state is in that level or in one of the next levels. Since the first automaton is in state s_1, and the third automaton is in state s_4, we see that the state is in the second level. The state of the second automaton in each level represents the correct branch. The second automaton is in state s_3, which means that the current state is in the right branch of the root. The input of the first flip-flop is identical to the input of the decision tree, but the input of all the other flip-flops is a function of the original input and the previous states.

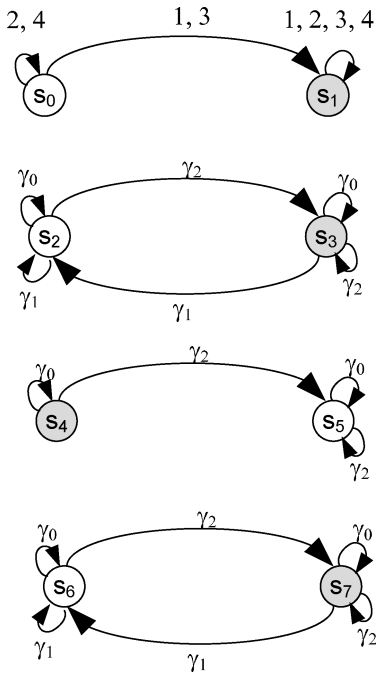

Figure 4: KR representation of the generalized decision tree.

We have so far shown that a generalized decision tree has a Krohn-Rhodes decomposition with $(d-1)(1+\log k)$ flip-flops. Our method to produce a tree of automata that can be privately computed requires $2^{d-1}2k$ nodes. For a balanced decision, with $d = \log n$, our solution requires $O(nk)$ nodes

5 Conclusions

In this work we tackle a new problem, that of se-

curing a distributed computation on inputs of *unbounded* length. Combined with the requirement of *non-interactivity* of the execution phase, this presents a setting where the large body of work on secure multiparty computation does not seem to be applicable. We define security in this setting, and show how to achieve it for a useful class of functions.

Our algorithms show the feasibility of a solution, and lay the foundations for future research in swarm computing, which may find applications in areas such as design of UAVs—unmanned aerial vehicles that collaborate in a common mission—as well as in cloud computing. Last but not least, our work puts forth an interesting application of the fundamental Krohn-Rhodes theory of automata decomposition.

Acknowledgments

It is a great pleasure to thank Azaria Paz and Stuart Margolis for useful discussions. We also thank the anonymous reviewers for *ICS 2011* for their helpful comments.

References

[1] M. Ben-OR, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[2] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively Secure Multi-Party Computation. In *STOC*, pages 639–648, 1996.

[3] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols(extended abstract). In *STOC*, pages 11–19, 1988.

[4] S. Dolev, J. Garay, N. Gilboa, and V. Kolesnikov. Swarming secrets. In *47th Annual Allerton Conference on Communication, Control, and Computing*, 2009.

[5] D. Dolev, C. Dwork, O. Waarts, and M. Yung. Perfectly secure message transmission. *J. ACM*, 40:1, pages 17–47, 1993.

[6] S. Dolev, L. Lahiani, and M. Yung. Secret swarm unit reactive k-secret sharing. In *INDOCRYPT*, pages 123–137, 2007.

[7] P. Domosi and C.L. Nehaniv. *Algebraic Theory of Automata Networks (SIAM Monographs on Discrete Mathematics and Applications, 11)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

[8] S. Eilenberg. *Automata, Languages, and Machines, Vol. B.* Academic Press, London, New York, NY, USA, 1976.

[9] C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *41st ACM Symposium on Theory of Computing (STOC)*, pages 169-178, 2009.

[10] F. Higgins, A. Tomlinson and K. Martin. Survey on Security Challenges for Swarm Robotics. *ICAS 2009*, pages 307–312.

[11] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *IEEE Globecom*, pages 99–102, 1987.

[12] K.R. Krohn and J. L. Rhodes. Algebraic theory of machines, 1962.

[13] K.R. Krohn and J. L. Rhodes. Algebraic theory of machines i: prime decomposition theorems for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, 1965.

[14] S. Margolis Complexity of holonomy decomposition. Private communication, February, 2010.

[15] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In *CCS '00: Proceedings of the 7th ACM conference on Computer and Communications Security*, pages 245–254, 2000.

[16] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC*, pages 73–85, 1989.

[17] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[18] M. Weiser. The Computer for the 21th Century. *Scientific American*, September, 1991.

[19] H.P. Zeiger. Cascade synthesis of finite state machines. *Information and Control*, 10:419–433, 1967. Erratum: Information and Control 11(4): 471 (1967).

A Proofs

We repeat the statements here for convenience.

Proposition 4.1. *Let $\mathcal{A} = (ST, \Gamma, \mu)$ be the automaton held by the dealer, and s_{init} an initial state. Fix a natural number k, and a sequence of k input symbols $(\gamma^1, \ldots, \gamma^k)$, $\gamma^j \in \Gamma$, $j = 1, \ldots, k$. Assume that executing $\mathcal{A}$ on $(\gamma^1, \ldots, \gamma^k)$ from state s_{init} terminates in state s_{term}. Then performing the initialization procedure described in Section 4.2, and all agents*

$P_1, \ldots, P_n$ executing Algorithm 1 on every γ^j and Algorithm 2 after γ^k is processed, returns the state s_{term}.

Proof: We prove the proposition by induction on k. The base case, $k = 0$, requires proving that immediately after initialization, executing Algorithm 2 returns s_{init}. By choice of the homomorphism $\Psi = (\Psi_1, \Psi_2)$, there exists a tuple $(s_1, \ldots, s_m)$ s.t. $\Psi_1(s_1, \ldots, s_m) = s_{init}$. The construction of Section 4.2 ensures that lines 2–9 of Algorithm 2 correctly determine $(s_1, \ldots, s_m)$. Since the dealer shares the secret $\Psi(s_1, \ldots, s_m)$ during initialization, Algorithm 2 returns s_{init}.

For the inductive step, assume that after all players receive the same input stream $(\gamma^1, \ldots, \gamma^j)$, execute Alg.1 on each input symbol, and run Alg.2 such that the output is the current state of $\mathcal{A}$ which we denote by s_{curr}. Assume that if the next input symbol is γ^{j+1}, then the next state of $\mathcal{A}$ is $s_{next} = \mu(s_{curr}, \gamma^{j+1})$. We prove that running Alg.1 on input γ^{j+1} and then running Alg.2 returns s_{next}.

By definition of the homomorphic representation we have a tuple $(s_1^c, \ldots, s_m^c)$ such that $\Psi_1(s_1^c, \ldots, s_m^c) = s_{curr}$. For all $\gamma \in \Gamma$ we have

$$\Psi_1(\mu_1(s_1^c, \varphi_1(\Psi_2^{-1}(\gamma^{j+1})), \ldots, \mu_m(s_m^c, \varphi_m(s_1^c, \ldots, s_m^c, \Psi_2^{-1}(\gamma^{j+1})))) = \mu(s_{curr}, \gamma^{j+1}) = s_{next}.$$

Since by induction, executing reconstruction by Algorithm 2 returns s_{curr}, there is a path of nodes $N_1, \ldots, N_m$ beginning at the root such that the secret value on the edge between N_i and N_{i+1} is 1, and for any other edge from N_i to level $i+1$ the secret value is 0. Furthermore, if N_i is a permutation automaton then the edge from N_i to N_{i+1} is marked with s_i^c, and otherwise (N_i is two flip-flops), the edge from N_i to N_{i+1} is marked with a flip-flop FF and the current state of FF is s_i^c.

In the online phase, Algorithm 1 goes over each node in $\mathcal{T}$, including N_1,...,N_m. For every $i = 1, \ldots, m$, if $\mathcal{A}_i$ is a permutation automaton then in line 12 the state marking the edge between N_i and N_{i+1} changes from s_i^c to $\mu_i(s, \varphi_i(s_1, \ldots, s_{i-1}, \Psi_2^{-1}(\gamma)))$. Since the share marking this edge does not change, the reconstruction algorithm collects $\mu_i(s, \varphi_i(s_1, \ldots, s_{i-1}, \Psi_2^{-1}(\gamma)))$ for $i = 1, \ldots, m$ in lines 2–9 and, thus, its output in line 10 is exactly s_{next}. □

Proposition 4.2. *The construction of Section 4.2—initialization procedure, and Algorithms 1 and 2—guarantees privacy in the progressive corruption model, according to Definition 3.3.*

Proof sketch: The privacy of our construction follows from the properties of secret sharing (cf. Definition 2.1) that we employ. It suffices to show that the adversary's view $\text{View}_\rho^\Pi(X, s)$ is distributed independently from ρ, X, and s, for a fixed length of ρ , say, $\ell \leq t$. In particular, the view of each execution is identical to the execution of $\mathcal{A}$ where agents $P_{i_1}, ..., P_{i_\ell}$ are corrupted simultaneously at the beginning of the execution. The latter claim is also easy to verify, as follows.

Firstly, we assume that the shares generated by different executions of secret-sharing are distributed identically. This easily achieved property is necessary, so that *Adv* would not be able to track the movement of shares in the progressive corruption model. Then the claim follows from the observation that each share of each state of each automaton in the tree is distributed identically at the time of the agent's corruption (given t or fewer corruptions). Further, all the labels of the children of each automaton node are a random permutation on the set of states of that node, and thus also do not carry any information about the state. □