ACM SIGACT News Distributed Computing Column 23

Sergio Rajsbaum*

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, Internet, and the Web. This issue consists of:

- "Want Scalable Computing? Speculate!" by Idit Keidar and Assaf Schuster
- "Security and Composition of Cryptographic Protocols: A Tutorial (Part I)" by Ran Canetti.

Many thanks to Idit, Assaf and Ran for their contributions to this issue.

I would like to bring your attention to a new book by a distinguished member of our distributed computing community, Gadi Taubenfeld: *Synchronization Algorithms and Concurrent Programming.* This is the first book to give a complete and coherent view of all aspects of synchronization algorithms in the context of concurrent programming. Dozens of algorithms are presented and their performance is analyzed according to precise complexity measures. For more details please go to: `http://www.faculty.idc.ac.il/gadi/book.htm`

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, open problems, and authors willing to write a guest column or to review an event related to theory of distributed computing.

Want Scalable Computing? Speculate!

Idit Keidar [1] and Assaf Schuster [2]

Abstract

Distributed computing is currently undergoing a paradigm shift, towards large-scale dynamic systems, where thousands of nodes collaboratively solve computational tasks. Examples of such emerging systems include autonomous sensor networks, data-grids, wireless mesh network (WMN) infrastructures, and more. We argue that *speculative computations* will be instrumental to successfully performing meaningful computations in such systems. Moreover, solutions deployed in such platforms will need to be as *local* as possible.

*Instituto de Matemáticas, UNAM. Ciudad Universitaria, Mexico City, D.F. 04510. `rajsbaum@math.unam.mx`.

[1]EE Department, Technion, Haifa 32000, Israel. `idish@ee.technion.ac.il`.

[2]CS Department, Technion, Haifa 32000, Israel. `assaf@cs.technion.ac.il`.

1 Introduction

Emerging distributed computing environments

Distributed systems of enormous scale have become a reality in recent years: Peer-to-peer file-sharing systems such as eMule [11] regularly report over a million online nodes. Grid computing systems like Condor [6] report harnessing tens of thousands of machines world-wide, sometimes thousands for the same application [29]. Publish-subscribe infrastructures such as Gryphon [27] and QuickSilver [26] can employ tens of thousands of machines to disseminate varied information to a large population of users with individual preferences, e.g., trading data to stock brokers, or update propagation in large-scale on-line servers. Wireless sensor networks are expected to span tens of thousands of nodes in years to come [8, 12]; testbeds of close to a thousand nodes are already deployed today [9, 7].

Nevertheless, in currently deployed large-scale distributed systems, nodes seldom collaboratively perform distributed computations or attempt to reach common decisions. The functionality of peer-to-peer and publish-subscribe systems is typically restricted to file searching and content distribution. Their topology does not adapt to global considerations, like network partitions or high traffic load at certain hot spots. Moreover, topic clustering, which has a crucial impact on the performance of publish-subscribe systems, is usually conducted offline, based on expected subscription patterns rather than continuously measured ones [30]. Computation grids serve mostly "embarrassingly parallel" computations, where there is virtually no interaction among the nodes. And current-day sensor networks typically only disseminate information (sometimes in aggregate form) to some central location [23, 12].

But this paradigm can be expected to change, as large scale systems performing non-trivial distributed computations will emerge. Simple file sharing applications are expected to evolve into large data-grids, in which thousands of nodes collaborate to provide complex information retrieval and data mining services [10, 28, 13]. As publish-subscribe systems will be used for larger and more diverse applications, they will need to optimize their topology and topic clustering adaptively.

Similar demands will arise in emerging wireless computing platforms, such as *mobile ad hoc networks (MANETs)*, *wireless mesh networks (WMNs)*, and sensor networks. In MANETs, a collection of mobile devices self-organize in order to allow communication among them in the absence of an infrastructure. Solving graph problems, like minimum vertex cover [15], is important for supporting efficient communication in such networks. In contrast to MANETs, a WMN provides an *infrastructure* for supporting communication among wireless devices, as well as from such devices to stationary nodes on the Internet. Such networks are expected to be deployed in city-wide scenarios, where potentially thousands of wireless routers will need to collectively engage in topology control, while dynamically adapting to varying loads. The WMN infrastructure will also need to dynamically assign Internet gateways to mobile devices that travel through the mesh. Since mobile devices are expected to run real-time rich-media applications, gateway assignment will need to maximize the quality-of-service (QoS). Hence, such assignments ought to take into account local considerations like proximity to the gateway, as well as global ones like load, since both considerations can impact the QoS. Optimizing a cost comprised of distance and load is called *load-distance balancing* [5]. For example, when the load (user population) and the set of servers (or gateways) are evenly divided across the network, the optimal cost is attained when each user is assigned to its nearest server. But when one area of the network is highly congested, a lower cost can be incurred by assigning some users from the loaded area to lightly-loaded remote servers.

Another example arises in sensor networks, which are expected to autonomously engage in complex decision making in a variety of application domains, ranging from detection of over-heating in data-centers, through disaster alerts during earthquakes, to biological habitat monitoring [24]. The nodes in such a network will need to perform a distributed computation in order to cooperatively compute some function of their inputs, such as testing whether the number of sensors reporting a problem exceeds a certain threshold,

or whether the average read exceeds another threshold. More generally, the nodes may need to compute some *aggregation function* of their inputs, e.g., majority vote, AND/OR, maximum, minimum, or average.

The need for local computing

Obviously, centralized solutions are not appropriate for such settings, for reasons such as load, communication costs, delays, and fault-tolerance. At the same time, traditional distributed computing solutions, which require global agreement or synchronization before producing an output, are also infeasible. In particular, in dynamic settings, where the protocol needs to be repeatedly invoked, each invocation entails global communication, inducing high latency and load. In fact, the latency for synchronization over a large WAN, as found in peer-to-peer systems and grids, can be so large that by the time synchronization is finally done, the network and data may well have already changed. Frequent changes may also lead to computing based on inconsistent snapshots of the system state. Moreover, synchronizing invocations that are initiated at multiple locations typically relies on a common sequencer (leader) [20], which by itself is difficult and costly to maintain.

Then how can one cope with the challenge of performing non-trivial computations in large-scale distributed systems? *Locality* is the key– there is a need for *local* solutions, whereby nodes make local decisions based on communication (or synchronization) with some proximate nodes, rather than the entire network.

Various different definitions of locality appear in the literature. But in general, a solution is typically said to be *local* if its costs (usually running time and communication) do not depend on the network size. A local solution thus has unlimited scalability. It allows for fast computing, low overhead, and low power consumption.

Speculation is the key to maximum locality

Unfortunately, although locality is clearly desirable, it is not always attainable. Virtually every interesting problem described above has some instances (perhaps pathological ones) that require global communication in order to reach the correct solution, even if most of its instances are amenable to local computation. Consider, for example, load-distance balancing in WMNs [5]. If the user load is evenly distributed across the entire WMN, then the optimal assignment, namely, assigning each user to its nearest gateway, can be computed locally by each router. But if load in one area of the network may be arbitrarily high, then the optimal solution may involve assigning users from this area to arbitrarily distant gateways, which may even involve communication with the entire network [5].

Locality has been considered in the context of distributed graph constructions, e.g., coloring [21, 25] and minimum vertex cover [15, 22], which are useful in MANETs. Such problems were mostly shown impossible to compute locally, (i.e., in constant time), except when limited to simplistic problems, approximations, or restricted graph models [21, 25, 16][3]. Another example arises in computing aggregation functions, such as majority voting, in sensor networks (or other large graphs) [4, 3, 2]. If the vote is a "landslide victory", e.g., when very few or almost all sensors report of a problem, then this fact can be observed in every neighborhood of the network. But in instances where the votes are at a near-tie, one must inevitably communicate with all nodes to discover the correct solution (if indeed an accurate solution is required).

Fortunately, in many real-life applications, practical instances are highly amenable to local computing [4, 16]. Furthermore, in many important areas, perfect precision is not essential. Consider, for instance, a query in a data mining application, where the top two results have approximately the same score. In such scenarios, it is often immaterial which of the two is returned.

[3] These local solutions typically operate in $O(\log^* n)$ time for a network of size n, which for practical purposes, can be seen as constant.

A desirable solution, therefore, should be "as local as possible" for each problem instance: In instances amenable to local computations, e.g., evenly distributed load in the WMN example, the system should converge to the correct result promptly. Moreover, it should become quiescent (stop sending messages) upon computing the correct result. In instances where reaching the correct solution requires communication with more distant nodes, the system's running times and message costs will inevitably be longer, but again, it is desirable that they be proportional to the distance that needs to be traversed in each instance.

Note, however, that each node by itself cannot deduce solely based upon local communication whether it is in a problem instance that can be solved locally or not. For example, a WMN node in a lightly-loaded area of the network cannot know whether some distant node will need to offload users to it. Therefore, in order to be "as local as possible", a solution must be *speculative*. That is, it must optimistically output the result that currently appears correct, even if this result may be later over-ruled because of messages that later arrive from distant nodes. In most instances, the computation will complete (and the system will become quiescent) a long time before any individual node will know that the computation is "done".

Fortunately, in the domains where such computations are applicable, it is not important for nodes to know when a computation is "done". This is because such platforms are constantly changing; that is, their computations are never "done". For example, inputs (sensor readings) in a sensor network change over time. Hence, the system's output, which is an aggregation function computed over these inputs, must also change. Regardless of whether speculation is employed, the user can never know whether the output is about to change, due to some recent input changes that are not yet reflected. The system can only ensure that once input changes cease, the output will eventually be correct. Using speculation provides the same semantics, but substantially expedites convergence in the majority of problem instances [2, 4]. At the same time, one should speculate responsibly: if the current output was obtained based on communication with 100 nodes, the algorithm shouldn't speculatively produce a different output after communicating with only 10 nodes.

Such dynamic behavior occurs in various large-scale settings: In data mining applications, the set of data items changes over time, as items are added, deleted, or modified. In topology control applications, nodes and communication links fail and get repaired. In WMNs, users join, depart, and move. Hence, speculative algorithms are useful in all of these platforms. Such algorithms can also be composed– for example, speculative majority voting can be used as a building block for a speculative data mining application [14].

To summarize, speculative computing is a promising approach for designing large-scale distributed systems. It is applicable to systems that satisfy the following conditions:

1. Global synchronization is prohibitive. That is, progress cannot be contingent on communication spanning the entire network.

2. Many problem instances are amenable to local solutions.

3. Eventual correctness is acceptable. That is, since the system is constantly changing, there isn't necessarily a meaningful notion of a "correct answer" at every point in time. But when the system stabilizes for "long enough", the output should converge to the correct one in the final configuration.

What does "as local as possible" mean?

The discussion above suggests that some problem instances are easier than others, or rather more amenable to local solutions. For some problems, one can intuitively recognize problem parameters that impact this amenability, for instance, the load discrepancies in load-distance balancing in WMNs, and the vote ratio in majority voting. Note that these problem parameters are *independent* of the system size– the same vote ratio can occur in a graph of any size. Suggested "local" solutions to these problems have indeed empirically exhibited a clear relationship between such parameters and the costs [4, 5]. However, how can this notion be formally defined?

The first formal treatment of "instance-based locality" was introduced by Kutten and Peleg in the context of local fault mending [18]. They linked the cost of a correction algorithm (invoked following faults) to the number of faults in an instance. Similar ideas were subsequently used in various studies of fault-tolerance [19, 17, 1]. In this context, the number of faults is the problem parameter that determines its amenability to local computation.

However, the problem parameter that renders instances locally computable is not always intuitive to pinpoint. Consider, for example, an algorithm computing *any* given aggregation function on a large graph. What is the problem parameter that governs its running time? In recent work with Birk et al. [3], we provide an answer to this question. We define a new formal metric on problem instances, *veracity radius (VR)*, which captures the inherent possibility to compute them locally. It is defined using the notion of an r-neighborhood of a node v, which is the set of all nodes within a radius r from v. Intuitively, if for all r-neighborhoods with $r \geq r_0$ the aggregation function yields the same value, then there is apparently no need for a node to communicate with nodes that are farther than r_0 hops away from it, irrespective of the graph size. The VR is then the minimum radius r_0 so that in all neighborhoods of radius $r \geq r_0$, the value of the aggregation function is the same as for the entire graph[4]. For example, if the aggregation function is majority voting, and the VR is 3, then the same value "wins" the vote in every 3-neighborhood in the graph. This value is clearly the majority over the entire graph, and every node can reach the globally correct result by communicating only with its 3-neighborhood. It is important to note that the nodes do not know the instance's VR, and hence cannot know that the outcome observed in the 3-neighborhood is the right one. However, a *speculative* algorithm can output the correct result once it gathers information within a radius of 3, and never change its output value thereafter.

Indeed, VR yields a tight lower bound on output-stabilization time, i.e., the time until all nodes fix their outputs to the value of the aggregation function, as well as an asymptotically tight lower bound on quiescence time [3]. Note that the output stabilization bound is for speculative algorithms: it is only seen by an external observer, whereas a node that runs the algorithm cannot know when it is reached. The above lower bounds are proven for input *classes* rather than individual instances. That is, for every given value r of the VR, no algorithm can achieve output stabilization or quiescence times shorter than r on *all* problem instances with a VR of r. In this context, the VR is the problem parameter that defines amenability to local computation. Empirical evaluation further shows that the performance of a number of efficient aggregation algorithms [31, 4, 14, 3] is effectively explained by the VR metric.

Although the VR metric is originally defined for a single problem instance, it has also been extended to deal with on-going aggregation over dynamically changing inputs [2]. This extension examines the VRs of all instances in a sliding window of input samples. As in the static case, this metric yields a lower bound on quiescence and output stabilization, and an algorithm whose performance is within a constant factor of the lower bound [2].

Discussion and additional research directions

This collection of examples naturally raises the question of whether the notion of "amenability to local computation" can be further generalized. An appealing way to try and generalize the notion of locality is by linking the per-instance performance of an algorithm directly to the performance of the optimal solution for this instance. For example, it would be desirable for an algorithm to perform within a factor of the best possible algorithm on every problem instance.

Unfortunately, it is only possible to prove such results in restricted special cases (e.g., load-distance balancing when restricted to a line instead of a plane [5]). There is no general way to do so, since it is often

[4]For practical purposes, the formal VR metric does not consider exact neighborhoods, but rather allows for some slack in the subgraphs over which the values of the aggregation function are examined [3].

not possible to design a single algorithm that is optimal for all instances. Consider again the question of aggregation on a graph. For every given (fixed) problem instance I, (that is, assignment I of input values to nodes), it is possible to design an algorithm as follows: each node locally checks whether its input matches its input in I. If yes, it speculatively decides on the value of the aggregation function on I, and does not send any messages. Otherwise, it initiates some well-known aggregation algorithm. If a node hears a message from a neighbor, it joins the well-known algorithm. This algorithm reaches quiescence and output stabilization within 0 time on instance I, which is clearly impossible for a single algorithm to achieve on all instances.

In conclusion, it remains a major challenge to find a general notion of "amenability to local computation" that will capture a wide range of distributed computations, including aggregation problems, fault mending, load-distance balancing, topic clustering in publish-subscribe systems, data mining, and so on. While such a general notion is still absent, appropriate metrics– and efficient speculative algorithms– for specific problems should continue to be sought. Of special interest will be on-going algorithms for dynamically changing systems, and algorithms providing *approximate* results rather than perfectly accurate ones.

Acknowledgments

We are thankful to Danny Dolev and Sergio Rajsbaum for helpful comments.

References

[1] Y. Azar, S. Kutten, and B. Patt-Shamir. Distributed error confinement. In *ACM Symp. on Prin. of Dist. Computing (PODC)*, July 2003.

[2] Y. Birk, I. Keidar, L. Liss, and A. Schuster. Efficient dynamic aggregation. In *Int'l Symp. on DIStributed Computing (DISC)*, Sept. 2006.

[3] Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff. Veracity radius - capturing the locality of distributed computations. In *ACM Symp. on Prin. of Dist. Computing (PODC)*, July 2006.

[4] Y. Birk, L. Liss, A. Schuster, and R. Wolff. A local algorithm for ad hoc majority voting via charge fusion. In *Int'l Symp. on DIStributed Computing (DISC)*, Oct. 2004.

[5] E. Bortnikov, I. Cidon, and I. Keidar. Load-distance balancing in large networks. Technical Report 587, Technion Department of Electrical Engineering, May 2006.

[6] Condor. High throughput computing. `http://www.cs.wisc.edu/condor/`.

[7] D. Culler. Largest tiny network yet. `http://webs.cs.berkeley.edu/800demo/`.

[8] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *4th Int'l Conf. on Information Processing in Sensor Networks (IPSN'05)*, Apr. 2005.

[9] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. In *5th Int'l Conf. on Information Processing in Sensor Networks (IPSN) Special track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, Apr. 2006.

[10] EGEE. Enabling grids for e-science. `http://public.eu-egee.org/`.

[11] eMule Inc. emule. `http://www.emule-project.net/`.

[12] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, pages 263–270, Aug. 1999.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, Oct. 2003.

[14] D. Krivitski, A. Schuster, and R. Wolff. A local facility location algorithm for sensor networks. In *Int'l Conf. on Distributed Computing in Sensor Systems (DCOSS)*, June 2006.

[15] F. Kuhn, T. Moscibroda, and R. Wattenhofer. What cannot be computed locally! In *ACM Symp. on Prin. of Dist. Computing (PODC)*, July 2004.

[16] F. Kuhn, T. Moscibroda, and R. Wattenhofer. On the locality of bounded growth. In *ACM Symp. on Prin. of Dist. Computing (PODC)*, July 2005.

[17] S. Kutten and B. Patt-Shamir. Time-adaptive self-stabilization. In *ACM Symp. on Prin. of Dist. Computing (PODC)*, pages 149–158, Aug. 1997.

[18] S. Kutten and D. Peleg. Fault-local distributed mending. In *ACM Symp. on Prin. of Dist. Computing (PODC)*, Aug. 1995.

[19] S. Kutten and D. Peleg. Tight fault-locality. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, Oct. 1995.

[20] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[21] N. Linial. Locality in distributed graph algorithms. *SIAM J. Computing*, 21:193–201, 1992.

[22] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1055, Nov. 1986.

[23] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *5th Symp. on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[24] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler. Wireless sensor networks for habitat monitoring. In *ACM Workshop on Sensor Networks and Applications*, Sept. 2002.

[25] M. Naor and L. Stockmeyer. What can be computed locally? *ACM Symp. on Theory of Computing (STOC)*, pages 184–193, 1993.

[26] K. Ostrowski and K. Birman. Extensible web services architecture for notification in large-scale systems. In *IEEE International Conference on Web Services (ICWS)*, 2006. To appear.

[27] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, and B. Mukherjee. Gryphon: An information flow based approach to message brokering. In *Fast Abstract in Int'l Symposium on Software Reliability Engineering*, Nov. 1998.

[28] TeraGrid. Teragrid project. `http://www.teragrid.org/`.

[29] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, Feb.–Apr. 2005.

[30] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *Parallel and Dist. Comp. and Systems (PDCS)*, Nov. 2005.

[31] R. Wolff and A. Schuster. Association rule mining in peer-to-peer systems. In *IEEE Int'l Conf. on Data Mining (ICDM)*, Nov. 2003.

Security and Composition of Cryptographic Protocols: A Tutorial (Part I)

Ran Canetti [5]

Abstract

What does it mean for a cryptographic protocol to be "secure"? Capturing the security requirements of cryptographic tasks in a meaningful way is a slippery business: On the one hand, we want security criteria that prevent "all potential attacks" against a protocol; on the other hand, we want our criteria not to be overly restrictive and accept "reasonable protocols". One of the main reasons for flaws is the often unexpected interactions among different protocol instances that run alongside each other in a composite system.

This tutorial studies a general methodology for defining security of cryptographic protocols. The methodology, often dubbed the "ideal model paradigm", allows for defining the security requirements of a large variety of cryptographic tasks in a unified and natural way. We first review more basic formulations that capture security in isolation from other protocol instances. Next we address the security problems associated with protocol composition, and review formulations that guarantee security even in composite systems.

1 Introduction

Cryptographic protocols, namely distributed algorithms that aim to guarantee some "security properties" in face of adversarial behavior, have become an integral part of our society and everyday lives. Indeed, we have grown accustomed to relying on the ubiquity and functionality of computer systems, whereas these systems make crucial use of cryptographic protocols to guarantee their "expected functionality." Furthermore, the security properties of cryptographic protocols and the functionality expected from applications that use them is being used by lawmakers to modify the ground rules of our society. It is thus crucial that we have sound understanding of how to specify, develop, and analyze cryptographic protocols.

The need for sound understanding is highlighted by the empirical fact that cryptographic protocols have been notoriously "hard to get right," with subtle flaws in protocols being discovered long after development, and in some cases even after deployment and standardization. In fact, even *specifying* the security properties required from protocols for a given task in a rigorous yet intuitively meaningful way has proved to be elusive.

The goal of this tutorial is to introduce the reader to the problems associated with formulating and asserting security properties of protocols, and to present a general methodology for modeling protocols and asserting their security properties. The tutorial attempts to be accessible to cryptographers and non-cryptographers alike. In particular, for the most part it assumes very little prior knowledge in cryptography. Also, while the main focus is on the foundational aspects of specifying security, the text attempts to be accessible and useful to practitioners as well as theoreticians. Indeed, the considered security concerns are realistic ones, and the end goal is to enable analyzing the security of real-world protocols and systems.

[5] IBM T.J. Watson Research Center, canetti@watson.ibm.com. Supported by NSF CyberTrust Grant #0430450.

Cryptographic tasks. In general, a cryptographic task, or a *protocol problem,* involves a set parties that wish to perform some joint computational task based on their respective local inputs, while guaranteeing some "security properties" in the face of various types of "adversarial behavior" by different components of the system and its users.

To get some feel for the range of issues and concerns involved, we briefly review some of the commonplace cryptographic tasks considered in the literature. Let us start with the very basic goal of guaranteeing secure communication between parties, in face of an external adversarial entity that has control over the communication network. Perhaps a first concern here is the *authenticity* of messages, namely finding out whether a received message indeed originates from its claimed sender, and whether it was modified en route. Another central concern is *secrecy,* namely guaranteeing that the adversary learns as little as possible on the communicated information. Additional concerns include anonymity, namely the ability to hide the identities of the communicating parties, and repudiation, namely the ability to prove to a third party that the communication took place. Salient tasks include key exchange, where two parties wish to agree on a random value (a key) that is known only to them; encryption, and digital signatures.

Another set of tasks, often called two-party tasks, involve two parties who are *mutually distrustful* but still wish to perform some joint computation. Here the only adversarial behavior under consideration is typically that of the parties themselves, and the communication medium is treated as trusted. Two main concerns are *correctness,* namely guaranteeing the validity of the local outputs as a function of the local inputs, and *privacy,* namely protecting the secrecy of local data as much as possible. This setting is discussed at length in the next section, where other concerns are mentioned. One such task is zero-knowledge (as in [GMRa89]), where one party wishes to convince the other in the correctness of some statement, without disclosing any additional information on top of the mere fact that the statement is correct. Another example is commitment, where a party C can *commit* to a secret value x, by providing some "commitment information" c that keeps x secret, while guaranteeing to a verifier that C can later come up with only one value x that's consistent with c. Another example is coin-tossing (as in, say, [B82]), namely the task where two parties want to agree on a bit, or a sequence of bits, that are taken from some predefined distribution, say the uniform distribution. This should hold even if one of the parties is trying to bias the output towards some value. (In fact, as discussed in later sections, coin-tossing involves some additional, implicit *secrecy* concerns regarding the public output. This is a good example of the subtleties involved in security modeling.) In addition to being natural tasks on their own, protocols for these tasks are often used as building blocks within protocols for obtaining more complex tasks.

A more general class is that of multi-party tasks where parties wish to perform some joint computation based on their private local data. Here, in addition to plain correctness and secrecy, there are other typically other task-specific concerns. We briefly mention some examples: Electronic voting, in a variety of contexts, require careful balancing among correctness, public accountability, privacy and deniability. Electronic-commerce applications such as on-line auctions, on line trading and stock markets, and plain on-line shopping require *fairness* in completion of the transaction, as well as the ability to resolve disputes in an acceptable way. On-line gambling tasks require, in addition, the ability to guarantee fair distribution of the outcomes. Privacy-preserving computations on databases introduce a host of additional concerns and goals, such as providing statistical information while preserving the privacy of individual entries, obtaining data while hiding from the database which data was obtained, and answering queries that depend on several databases without leaking additional information in the process. Secure distributed depositories, either via a centrally-managed distributed system or in an open, peer-to-peer fashion, involve a host of additional secrecy, anonymity, availability and integrity concerns.

Defining security of protocols. There is vast literature describing protocols aimed at solving the problems mentioned above, and many others, in a variety of settings. Out of this literature, let us mention only the works of Yao [Y86], and Goldreich, Micali and Wigderson [GMW87], which give a mechanical way to generate protocols for solving practically *any* multi-party cryptographic protocol problem "in a secure way", assuming authenticated communication. (These constructions do not cover all tasks; for instance the do not address the important problem of *obtaining* authenticated communication. Still, they are very general.)

But, what does it mean for a cryptographic protocol to solve a given protocol problem, or a cryptographic task, "in a secure way"? How can we formalize the relevant security requirements in a way that makes mathematical sense, matches our intuition, and at the same time can also be met by actual protocols? This turns out to be a tricky business.

Initially, definitions of security were problem-specific. That is, researchers came up with ad-hoc modeling of protocols and a set of requirements that seems to match the intuitive perception of the problem at hand. In addition, definitions were often tailored to capture the properties of specific solutions or protocols. However, as pointed out already in [Y82A], we would like to have a general framework for specifying the security properties of different tasks. A general framework allows for more uniform and methodological specification of security properties. Such a specification methodology may provide better understanding of requirements and their formalization. It is also likely to result in fewer flaws in formulating the security requirements of tasks.

There is also another argument in favor of having a general analytical framework. Traditionally, notions of security tend to be very sensitive to the specific "execution environment" in which the protocol is running, and in particular to the other protocols running in the system at the same time. Thus, a set of requirements that seem appropriate in one setting may easily become completely insufficient when the setting is changed only slightly. This is a serious drawback when trying to build secure systems that make use of cryptographic protocols. Here a general analytical framework with a uniform methodology of specifying security requirements can be very useful: It allows formulating statements such as "Any protocol that realizes some task can be used in conjunction with any protocol that makes use of this task, without bad interactions," or "Protocols that realize this task continue to realize it in any execution environment, regardless of what other protocols run in the system." Such *security-preserving composition* theorems are essential for building security protocols in a systematic way. They can be meaningful only in the context of a general framework for representing cryptographic protocols.

Several general frameworks for representing cryptographic protocols and specifying the security requirements of tasks were developed over the years, e.g. [GL90, MR91, B91, BCG93, PW94, C00, HM00, DM00, PW00, MRST01, C01]. These frameworks differ greatly in their expressibility (i.e., the range of security concerns and tasks that can be captured), in the computational models addressed, and in many significant technical details. They also support different types of security-preserving composition theorems. Still, all of these frameworks follow in one way or another the same underlying definitional approach, called the *trusted-party paradigm.*

This tutorial. This tutorial concentrates on the trusted-party definitional paradigm and the security it provides. Special attention is given to the importance of security-preserving composition in cryptographic protocols, and to the composability properties of this paradigm in particular. We also provide some comparisons with other (non-cryptographic) general approaches for modeling distributed protocols and analyzing their properties. For sake of concreteness, we concentrate on two specific formalizations of the trusted-party paradigm. The first one, based on [C00], is somewhat simpler and provides a rather basic notion of security, with some limited form of security-preserving composition. The second one is that of [C01], called *universally composable (UC) security.* This framework is more expressive and provides stronger composability

guarantees, at the price of more stringent requirements and a somewhat more complex formalism.

Throughout, the presentation attempts to balance readability and rigor. In particular, we try to keep the presentation as informal as possible, while providing enough details so as to allow for unambiguous extraction of a formal model. We also try to highlight the salient points out of the many inevitable details. We also try to introduce the definition in a gradual way, at the price of a somewhat more lengthy presentation.

This is the first part of a two-part tutorial. This part concentrates on the basic notion of security, and its formalization for the "stand-alone" case where only a single execution of a protocol is considered, in isolation from other executions. The second part concentrates on the concerns that arise from running multiple protocols in the same system. It also includes a mini-survey of definitions and formalizations of secure protocols.

Section 2 presents and motivates the general paradigm used to define security of protocols. Section 3 presents a simplified formalization of the general paradigm. While considerably restricted in its expressibility, this formulation allows concentrating on the main ideas without much of the complexity of the general case. Section 4 presents a more general formulation, while still concentrating on the stand-alone case. It also briefly discusses some basic general feasibility results. Many relevant works are mentioned as we go along. However, due to lack of space, a more organized survey or related work is deferred to the second part of this tutorial.

2 The trusted-party paradigm

This section motivates and sketches the trusted-party definitional paradigm, and highlights some of its main advantages. More detailed descriptions of actual definitions are left to subsequent sections.

Let us consider, as a generic example, the task of two-party secure function evaluation. Here two mutually distrustful parties P_0 and P_1 want to "securely evaluate" some known function $f : D^2 \to D$, in the sense that P_i has value x_i and the parties wish to jointly compute $f(x_0, x_1)$ "in a secure way." Which protocols should we consider "secure" for this task?

First attempts. Two basic types of requirements come to mind. The first is correctness: the parties that follow the protocol (often called the "good parties") should output the correct value of the function evaluated at the inputs of all parties. Here the "correct function value" may capture multiple concerns, including authenticity of the identities of the participants, integrity of the input values, correct use of random choices, etc. The second requirement is secrecy, or hiding the local information held by the parties as much as possible. Here it seems reasonable to require that parties should be able to learn from participating in the protocol nothing more than their prescribed outputs of the computation, namely the "correct" function value. For instance, in the database example from the beginning of the introduction, correctness means that the parties output all the entries which appear in both databases, and only those entries. Privacy means that no party learns anything from the interaction *other than* the joint entries.

However, formalizing these requirements in a meaningful way seems problematic. Let us briefly mention some of the problematic issues. First, defining correctness is complicated by the fact that it is not clear how to define the "input value" that an arbitrarily-behaving party contributes to the computation. In particular, it is of course impossible to "force" such parties to use some value given from above. So, what would be a "legitimate", or "acceptable" process for choosing inputs by parties who do not necessarily follow the protocol?

Another question is how to formulate the secrecy requirement. It seems natural to have a definition based either on some sort of indistinguishability between two distributions, or alternatively on some notion of "simulation" of the adversary's view as in the case of probabilistic encryption or zero-knowledge [GM84,

GMRa89, G01]. But it is not clear at this point what should the two distributions be, or in what setting the "simulator" should operate. In particular, how should the fact that the adversary inevitably obtains the "correct function value" be incorporated in the secrecy requirement?

In fact, at a second look the correctness and secrecy requirements seem inherently "intertwined" in a way that prevents separating them out as different requirements. Already from the above discussion it is apparent that the secrecy requirement must somehow depend on the definition of the "correct function value". In addition, as demonstrated by the following example, the definition of "correct function value" must in itself depend on some secrecy requirement.

Assume that $x_0, x_1 \in \{0,1\}$, and that f is the exclusive or function, namely $f(x_0, x_1) = x_0 \oplus x_1$. The protocol instructs P_0 to send its input to P_1; then P_1 announces the result. Intuitively, this protocol is insecure since P_1 can unilaterally determine the output, *after learning P_0's input.* Yet, the protocol maintains secrecy (which holds vacuously for this problem since each party can infer the input of the other party from its own input and the function value), and is certainly "correct" in the sense that the output fits the input that P_1 "contributes" to the computation.

This example (taken from [MR91]) is instructive in more than one way. First, it highlights the fact that correctness and secrecy requirements depend on each other in a seemingly circular way. Second, it brings forth another security requirement from protocols, in addition to correctness and secrecy: We want to prevent one party from *influencing* the function value in illegitimate ways, even when plain correctness is not violated.

Additional problems arise when the function to be evaluated is *probabilistic,* namely the parties wish to jointly "sample" from a given distribution that may depend on secret values held by the parties. Here it seems clear that correctness should take the form of some statistical requirement from the output distribution. In particular, each party should be able to influence the output distribution only to the extent that the function allows, namely only in ways that can be interpreted as providing a legitimately determined input to the function. Furthermore, as demonstrated by the following example, the case of probabilistic functions puts forth an additional, implicit secrecy requirement.

Assume that the parties want to toss k coins, where k is a security parameter; formally, the evaluated function is $f(\cdot,\cdot) = r$, where $r \stackrel{R}{\leftarrow} \{0,1\}^k$. Furthermore, for simplicity assume that we trust the parties to follow the protocol instructions (but still want to prevent illegitimate information leakage). Let f be a one-way permutation on domain $\{0,1\}^k$. The protocol instructs P_0 to choose $s \stackrel{R}{\leftarrow} \{0,1\}^k$ and send $r = f(s)$ to P_1. Both parties output r.

This protocol preserves secrecy vacuously (since the parties do not have any secret inputs), and is also perfectly correct in the sense that the distribution of the joint output is perfectly uniform. However, the protocol lets P_0 hold some "secret trapdoor information" on the joint random string. Furthermore, P_1 does not have this information, and cannot feasibly compute it (assuming that f is one-way). As we will see, this "quirk" of the protocol is not merely an aesthetic concern. Having such trapdoor information can be devastating for security if the joint string r is used within other protocols.

Other concerns, not discussed here, include issues of *fairness* in obtaining the outputs (namely, preventing parties from aborting the computation after they received their outputs but before other parties received theirs), and addressing *break-ins* into parties that occur during the course of the computation.

The above discussion seems to suggest that it may be better to formulate a *single,* unified security requirement, rather than making several separate ones. This requirement would appropriately intertwine the various correctness, secrecy, and influence concerns. Furthermore, to sidestep the apparent circularity in the requirements, and to prevent implicit information leaks as in the coin-tossing example, it seems that this requirement should somehow specify also the *process* in which the output is to be obtained.

The trusted party paradigm. The trusted party paradigm follows the "unified requirement" approach. The idea (which originates in [GMW87], albeit very informally) proceeds as follows. In order to determine whether a given protocol is secure for some cryptographic task, first envision an ideal process for carrying out the task in a secure way. In the ideal process all parties secretly hand their inputs to an external trusted party who locally computes the outputs according to the specification, and secretly hands each party its prescribed outputs. This ideal process can be regarded as a "formal specification" of the security requirements of the task. (For instance, to capture the above *secure function evaluation* task, the trusted party simply evaluates the function on the inputs provided by the parties, and hands the outputs back to the parties. If the function is probabilistic then the trusted party also makes the necessary random choices.) The protocol is said to securely realize a task if running the protocol amounts to "emulating" the ideal process for the task, in the sense that any damage that can be caused by an adversary interacting with the protocol can also be caused by an adversary in the ideal process for the task.

A priori, this approach seems to have the potential to capture all that we want, and in particular untangle the circularity discussed above. Indeed, in the ideal process both correctness and lack of influence are guaranteed *in fiat,* since the inputs provided by any adversarial set of parties cannot depend on the inputs provided by the other parties in any way, and furthermore all parties obtain the correct output value according to the specification. Secrecy is also immediately guaranteed, since the only information obtained by any adversarial coalition of parties is the legitimate outputs of the parties in this coalition. In particular, no implicit leakage of side-information correlated with the output is possible. Another attractive property of this approach is its apparent generality: It seems possible to capture the requirements of very different tasks by considering different sets of instructions for the external trusted party.

It remains to formalize this definitional approach in a way that maintains its intuitive appeal and security guarantees, and at the same time allows for reasonable analysis of "natural" protocols. In this tutorial we describe several formalizations, that differ in their complexity, generality and secure composability guarantees. Yet, all these formalizations follow the same outline, sketched as follows. The definition proceeds in three steps. First we formalize the process of executing a distributed protocol in the presence of adversarial behavior of some parts of the system. Here the adversarial behavior is embodied via a single, centralized computational entity called an adversary. Next we formalize the ideal process for the task at hand. The formalized ideal process also involves an adversary, but this adversary is rather limited and its influence on the computation is tightly controlled. Finally, we say that a protocol π securely realizes a task $\mathcal{F}$ if for *any* adversary $\mathcal{A}$ that interacts with π there *exists* an adversary $\mathcal{S}$ that interacts with the trusted party for $\mathcal{F}$, such that no "external environment," that gives inputs to the parties and reads their outputs, can tell whether it is interacting with π or with the trusted party for $\mathcal{F}$.

Very informally, the goal of the above requirement is to guarantee that any information gathered by the adversary $\mathcal{A}$ when interacting with π, as well as any "damage" caused by $\mathcal{A}$, could have also been gathered or caused by an adversary $\mathcal{S}$ in the ideal process with $\mathcal{F}$. Now, since the ideal process is designed so that *no* $\mathcal{S}$ can gather information or cause damage more than what is explicitly permitted in the ideal process for $\mathcal{F}$, we can conclude that $\mathcal{A}$ too, when interacting with π, cannot gather information or cause damage more than what is explicitly permitted by $\mathcal{F}$. Another attractive property of this formulation is its apparent "compositionality": Since it is explicitly required that no "environment" can tell the protocol from the trusted party, it makes sense to expect that a protocol will exhibit the same properties regardless of the activity in the rest of the system.

3 Basic security: A simplified case

For the first formalization, we consider a relatively simple setting: As in the previous section, we restrict ourselves to two-party secure function evaluation, namely the case of two parties that wish to jointly compute a function of their inputs. We also restrict ourselves to the "stand-alone" case, where the protocol is executed once, and no other parties and no other protocol executions are considered. Furthermore, we are only concerned with the case where one of the two parties is adversarial. In particular, the communication links are considered "trusted", in the sense that each party receives all the messages sent by the other party, and only those messages. It turns out that this setting, while highly simplified, actually captures much of the complexity of the general problem. We thus present it in detail before presenting more general (and more realistic) settings.

Section 3.2 presents the definition. Section 3.3 exemplifies the definition by providing some definitions of cryptographic tasks, cast in this model. First, however, we present the underlying model of distributed computation, in Section 3.1.

3.1 A basic system model

Before defining security of protocols, one should first formulate a model for representing distributed systems and protocols within them. To be viable, the model should enable expressing realistic communication media and patterns, as well as realistic adversarial capabilities. This section sketches such a model. Readers that are satisfied with a more informal notion of distributed systems, protocols, and polynomial-time computation can safely skip this section.

Several general models for representing and arguing about distributed systems exist, e.g. the CSP model of Hoare [H85], the π-calculus of Milner [M89, M99], or the I/O automata of Lynch [Ly96]. Here we use the interactive Turing machines (ITMs) model, put forth in an initial form in Goldwasser, Micali and Rackoff [GMRa89] (see also [G01]). Indeed, while the ITM model is more "low level" and provides fewer and less elegant abstraction mechanisms than the above models, it allows for capturing in a natural way the subtle relations between randomization, interaction, and resource-bounded adversarial behavior. Specifically, we formulate a simplified version of the model of [C01], that suffices for the two-party, stand-alone case. A richer model, that's more appropriate for studying general systems, is mentioned in Section 4.1. (As expected, many details of this model are to a large extent arbitrary, in the sense that other choices would have been equally good for our needs. Yet, for concreteness we need to pin down some specific model.)

Interactive Turing Machines. An ITM is a Turing machine with some "shared tapes" which can be written into by one machine and read by another. Specifically, we use three externally writable tapes, namely tapes that can be written on by other ITMs: an input tape, representing inputs provided by the "invoking program", an incoming communication tape, representing messages coming from the network, and a subroutine output tape, representing outputs provided by subroutines invoked by the present program. The distinction between these ways of providing information to a running program is instrumental for modeling security. In particular, The input tape represents information coming from "outside the protocol execution", while the incoming communication tape and the subroutine output tapes provide information that is "internal to a protocol execution." Also, the incoming communication tape models information coming from untrusted sources, while the information on the subroutine output tapes is treated as coming from a trusted source. Finally, we concentrate on *probabilistic* machines, namely on machines that can make random choices (or, equivalently, machines that have random tapes.)

Systems of ITMs. The model of computation consists of several instances of ITMs that can write to the externally writable tapes of each other, subject to some global rules. We call an ITM instance an ITI. Different ITIs can run the same code (ITM); however they would, in general, have different local states.

An execution of a systems of ITMs consists of a sequence of activations of ITIs. In each activation, the active ITI proceeds according to its current state and contents of tapes until it enters a special wait state. In order to allow the writing ITI to specify the target ITI we enumerate the ITIs in the system in some arbitrary order, and require that the write instruction specify the numeral of the target ITI. (This addressing mechanism essentially assumes that each two ITIs in the system have a "direct link" between them. A more general addressing mechanism is described in Section 4.1.) In order to simplify determining the order of activations, we allow an ITI to write to an externally writable tape of at most one other ITI per invocation. The order of activation is determined as follows: There is a pre-determined ITI, called the initial ITI, which is the first one to be activated. At the end of each activation, the ITI whose tape was written to is activated next. If no external write operation was made then the initial ITI is activated. The execution ends when the initial ITI halts.

In principle, the global input of an execution should be the initial inputs of all ITIs. For simplicity, however, we define the global input as the input of the initial ITI alone. Similarly, the output of an execution is the output of the initial ITI.

A final ingredient of a system of ITMs is the control function, which determines which tapes of which ITI can each ITI write to. As we'll see, the control function will be instrumental in defining different notions of security.

Looking ahead, we remark that this very rudimentary model of communication, with its simple and sequential scheduling of events, actually proves sufficient for expressing general synchrony, concurrency, and scheduling concerns.

Polynomial-Time ITMs. In order to model resource-bounded programs and adversaries, we need to define resource-bounded ITMs. We concentrate on polynomial time ITMs. We wish to stick with the traditional interpretation of polynomial time as "polynomial in the length of the input." However, since in our model ITMs can write to the tapes of each other, care should be taken to guarantee that the overall running time of the system remains polynomial in the initial parameters. We thus say that an ITM M is polynomial time (PT) if there exists a polynomial $p(\cdot)$ such that at any point during the computation the overall number of steps taken by M is at most $p(n)$, where n is the overall number of bits written so far into the *input* tape of M, minus the number of bits written by M to the input tapes of other ITIs. This guarantees that a system of communicating ITMs completes in polynomial time in the overall length of inputs, even when ITIs write to the input tapes of each other. (An alternative, somewhat simpler formulation says that the overall running time of an ITM should be polynomial in the value of a "security parameter". However, this requirement considerably limits the expressibility of the model, especially in the case of reactive computation.)

Protocols. A protocol is defined simply as an ITM. This ITM represents the code to be run by each participant, namely the the set of instructions to be carried out upon receipt of an input, incoming message, or subroutine output (namely, output from a subroutine). If the protocol has different instructions for different roles, then the ITM representing the protocol should specify the behaviors of all roles. A protocol is PT if it is PT as an ITM.

3.2 The definition of security

We flesh out the definitional plan from Section 2, for the case of two-party, stand-alone, non-reactive tasks.

The protocol execution experiment.[6] Let π be a two-party protocol. The protocol execution experiment proceeds as follows. There are three entities (modeled as ITIs): An ITI, called P, that runs the code of π, an ITI called the adversary, denoted $\mathcal{A}$, and an ITI called the environment, denoted $\mathcal{E}$.

The environment $\mathcal{E}$ is the initial ITI; thus it is activated first. Its role in the interaction is limited: All it does is provide initial inputs to $\mathcal{A}$ and the party P running π, and later obtains their final outputs. (The initial inputs can be thought of as encoded in $\mathcal{E}$'s own input.)

Once either P or $\mathcal{A}$ is activated, with either an input value or an incoming message (i.e., a value written on the incoming communication tape), it runs its code and potentially generates an output (to be read by $\mathcal{E}$), or a message to be written on the other party's incoming communication tape.

The final output of the execution is the output of the environment. As we'll see, it's enough to let this output consist of s single bit.

We use the following notation. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)$ denote the random variable describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and protocol π on input x (for $\mathcal{E}$). Here the probability is taken over the random choices of all the participating ITIs. Let $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ denote the ensemble of distributions $\{\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}(x)\}_{x\in\{0,1\}^*}$.

The ideal process. Next an ideal process for two-party function evaluation is formulated. Let $f : (\{0,1\}^*)^2 \to (\{0,1\}^*)^2$ be the (potentially probabilistic) two-party function to be evaluated. Here the ith party, $i \in \{1,2\}$, has input $x_i \in \{0,1\}^*$, and needs to obtain the ith coordinate in $f(x_1, x_2)$.

We want to formalize a process where the parties hand their inputs to a trusted entity which evaluates f on the provided inputs and hands each party its prescribed output. For that purpose, we add to the system an additional ITI, denoted $\mathcal{T}_f$, which represents the trusted party and captures the desired functionality. The ITI P now runs the following simple ideal protocol for f: When receiving input value, P copies this value to the input tape of $\mathcal{T}_f$. When receiving an output from $\mathcal{T}_f$ (on its subroutine output tape), P copies this output to the subroutine output of $\mathcal{E}$. Messages on the incoming communication tape are ignored.

$\mathcal{T}_f$ proceeds as follows: It first waits to receive input (b, x) from P and input x' from the adversary $\mathcal{A}$, where $b \in \{1,2\}$ denotes whether x is to be taken as the first or second input to f. Once the inputs are received, $\mathcal{T}_f$ lets $x_b \leftarrow x$, $x_{3-b} \leftarrow x'$, and $(y_1, y_2) \leftarrow f(x_1, x_2)$. Next, $\mathcal{T}_f$ outputs y_{3-b} to $\mathcal{A}$. Once it receives an $\texttt{ok}$ message from $\mathcal{A}$, $\mathcal{T}_f$ outputs y_b to P.

Analogously to the protocol execution experiment, let $\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}(x)$ denote the random variable describing the output of environment $\mathcal{E}$ when interacting with adversary $\mathcal{A}$ and the ideal protocol for f on input x (for $\mathcal{E}$), where the probability is taken over the random choices of all the participating ITIs. Let $\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}$ denote the ensemble $\{\text{IDEAL}_{f,\mathcal{A},\mathcal{E}}(x)\}_{x\in\{0,1\}^*}$.

Securely evaluating a function. Essentially, a two-party protocol π is said to securely evaluate a two-party function f if for *any* adversary $\mathcal{A}$, that interacts with π, there *exists* another adversary, denoted $\mathcal{S}$, that interacts with $\mathcal{T}_f$, such that no environment will be able to tell whether it is interacting with π and $\mathcal{A}$, or alternatively with $\mathcal{T}_f$ and $\mathcal{S}$.

To provide a more rigorous definition, we first define indistinguishability of probability ensembles. A function is negligible if it tends to zero faster than any polynomial fraction, when its argument tends to infinity. Two distribution ensembles $\mathcal{X} = \{X_i\}_{a\in\{0,1\}^*}$ and $\mathcal{X}' = \{X'_i\}_{a\in\{0,1\}^*}$ are indistinguishable (denoted $\mathcal{X} \approx \mathcal{X}'$) if for any $a, a' \in \{0,1\}^k$ the statistical distance between distributions X_a and X'_a is a negligible function of k. (The use of an asymptotic notion of similarity between distribution ensembles greatly

[6] The presentation below is somewhat informal. Formal description, in terms of a system of ITMs as sketched in the previous section, can be easily inferred. In particular, the various model restrictions are enforced via an appropriate control function.

simplifies the presentation and argumentation. However it inevitably introduces some slack in measuring distance. More precise and quantitative notions of similarity may be needed to determine the exact quantitative security of protocols. Also, note that we do not define *computational* indistinguishability of probability ensembles. This is so since we will only be interested in ensembles of distributions over the binary domain $\{0,1\}$, and for these ensembles the two notions are equivalent.) Secure evaluation is then defined as follows:

Definition 1 (Basic security for two-party function evaluation) *A two-party protocol π* securely evaluates *a two-party function f if for any PT adversary $\mathcal{A}$ there exists a PT adversary $\mathcal{S}$ such that for all PT environments $\mathcal{E}$ that output only one bit:*

$$\text{IDEAL}_{f,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \tag{1}$$

3.2.1 Discussion

Motivating some choices in the model. Recall that the protocol execution experiment involves only a *single* party running the two-party protocol, where the messages are exchanged with the adversary rather than with another party running the protocol. This models the fact that we consider the behavior of the system where one of the parties follows the protocol while the other follows a potentially different strategy. In two-party protocols where there are two distinct roles there will be two distinct cases depending on the role played by the party who is running the protocol. However, since the role can be modeled as part of the input, this distinction need not be made within the general modeling.

The environment captures the "external system" that provides inputs to the parties and obtains their outputs. In particular, it chooses an input value to the party P running the protocol, and some (potentially correlated) initial information to the adversary. The environment also obtains the final outputs, both of P and of the adversary. In other words, the environment essentially sees the "I/O behavior", or "functionality" of the protocol and its adversary, without having access to the communication between the parties.

In the present simplified case, an alternative formulation of the model that replaces the environment with an "external entity" that merely looks at the outcome of the experiment is equivalent to the present formulation. We opt to model the environment as an entity that takes part in the actual execution experiment in order to be compatible with the more general formalization presented in the next section. Indeed, there it is important to allow the environment to be active *during* the protocol execution experiment.

The ideal process represents in a straightforward way the intuitive notion of a trusted party that obtains the inputs from the parties and locally computes the desired outputs. In particular, the input provided by the adversary depends only in the information it was initially given from $\mathcal{E}$ (which may or may not be correlated with the input given to P). Furthermore, $\mathcal{A}$ obtains only the specified function value. Yet, the present formulation of the ideal process does not guarantee *fairness:* $\mathcal{A}$ always receives the output first, and can then decide whether P will obtain its output.

Interpreting the definition. It is instructive to see how the informal description of Section 2 is implemented. Indeed, if there existed an adversary $\mathcal{A}$ that could interact with the protocol and exhibit "bad behavior" that cannot be exhibited in the ideal process, by any adversary $\mathcal{S}$, then there would exist an environment $\mathcal{E}$ that outputs '1' with significantly different probabilities in the two executions, and the definition would be violated.

The present formulation interprets "bad behavior" in a very broad sense, namely in terms of the joint distribution of the outputs of P and $\mathcal{A}$ on any given input. This allows testing, e.g., whether the protocol allows the adversary to gather information on the other party's input, where this information is not available in the ideal process; it also allows testing whether the protocol allows the adversary to *influence* the output

of the other party in ways that are not possible in the ideal process. In particular, it is guaranteed that the adversary $\mathcal{S}$ in the ideal process is able to generate an "effective adversarial input" x_2 to the trusted party that is consistent with P's input and output (namely, x_2 satisfies $y_1 = f(x1, x2)_1$, where x_1 is P s input and y_1 is P's output).

In addition, the environment can choose to provide $\mathcal{A}$ with input that is either uncorrelated with P's input, or alternatively partially or fully correlated with P's input. This guarantees that the the above properties of the protocol hold regardless of how much "partial information" on P's input has leaked to the adversary beforehand.

Also, notice that the correctness requirement has a somewhat different flavor for deterministic and probabilistic functions: For deterministic functions, the correctness is absolute, in the sense that the output of the parties that follow the protocol is guaranteed to be the exact function value. For probabilistic functions, it is only guaranteed that the outputs are *computationally indistinguishable* from the distribution specified by the function. This difference allows the analyst to choose which level of security to require, by specifying an appropriate f.

Extensions. The definition can be modified in natural ways to require an information-theoretic level of security, by considering computationally unbounded adversaries and environments, or even *perfect* security, by requiring in addition that the two sides of (1) be identical. (To preserve meaningfulness, $\mathcal{S}$ should still be polynomial in the complexity of $\mathcal{A}$, even when $\mathcal{A}$ and $\mathcal{E}$ are unbounded.)

Similarly, the definition can be modified to consider only restricted types of malicious behavior of the parties, by appropriately restricting the adversary. In particular, security against "semi-honest" parties that follow the protocol, but may still try to gather additional information, can be captured by requiring $\mathcal{A}$ to follow the original protocol.

Another, more technical issue has to do with the order of quantifiers: A priori, it may seem that the above order of quantifiers is too restrictive, and a formulation where $\mathcal{S}$ may depend on $\mathcal{E}$ would suffice for guaranteeing basic security. It turns out, though, that the two formulations are equivalent (the proof is similar to that in [C01]). We use the present formulation since it seems more natural, and furthermore it simplifies proving composability results, discussed in later sections.

Secure evaluation vs. observational equivalence. Finally, we compare this definition of security with the notion of *observational equivalence* of Milner [M89, M99], used in the π-calculus formalism and elsewhere. (This notion is sometimes called also *bi-simulatability.*) The two notions have somewhat of the same flavor, in the sense that both notions require that an external environment (or, *context*) will be unable to tell whether it is interacting with one process or with another. However, the present notion is significantly weaker, since it allows the additional "leeway" of constructing an appropriate simulator $\mathcal{S}$ that will help "fool" the external environment. This extra "intentional weakness" of the present notion is in fact the core of what makes it realizable for interesting cryptographic tasks, while maintaining much of the meaningfulness. Another difference is that the present notion is not symmetric, whereas observational equivalence is. We note that the more general formulations of the notion of security, presented in subsequent sections, appear even closer in spirit to observational equivalence. Still, the main difference is the same as here.

3.3 Examples

To exemplify the use of Definition 1 for capturing the security requirements of cryptographic tasks, we use it to capture the security requirements of three quite different tasks. In fact, all that remains to be done in order to define protocols that realize a task is to formulate an appropriate two-party function:

Zero Knowledge. Let $R : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ be a binary relation, and consider the bivariate function $f^R_{\text{ZK}}((x,w),-) = (-,(x,R(x,w)))$. That is, the first party (the "prover") has input (x,w), while the second party (the "verifier") has empty input. The verifier should learn x plus the one-bit value $R(x,w)$, and nothing else. The prover should learn nothing from the interaction. In particular, when R is the relation associated with an NP language L (that is, $L = L_R \stackrel{\text{def}}{=} \{x|\exists w \text{ s.t. } R(x,w) = 1\}$), these requirements are very reminiscent of the requirements from a Zero-Knowledge protocol for L: The verifier is guaranteed that it accepts, or outputs $(x,1)$, only when $x \in L$ (soundness), and the prover is guaranteed that the verifier learns nothing more other than whether $x \in L$ (zero-knowledge).

It might be tempting to conclude that a protocol is Zero-Knowledge for language L_R as in [GMRa89] if and only if it securely realizes f^R_{ZK}. This statement is true "in spirit", but some caveats exist. First, [GMRa89] define Zero Knowledge so that both parties receive x as input, whereas here the verifier learns x only via the protocol. This difference, however, is only "cosmetic" and can be resolved via simple syntactic transformations between protocols.

The remaining two differences are more substantial: First, securely realizing f^R_{ZK} only guarantees "computational soundness", namely soundness against PT adversarial provers. Second, securely realizing f^R_{ZK} implies an additional, somewhat implicit requirement: When the adversary plays the role of a potentially misbehaving prover, the definition requires the simulator to explicitly hand the input x *and the witness* w to the trusted party. To do this, the simulator should be able to "extract" these values from the messages sent by the adversary. This requirement has the flavor of a *proof of knowledge* (see e.g. [G01]), albeit in a slightly milder form that does not require a *black-box* extractor.

In conclusion, we have that a protocol securely realizes f^R_{ZK} if and only if a slight modification of the protocol is a computationally sound Zero-Knowledge Proof of Knowledge for L_R (with potentially non black-box extractors).

Database Intersection. As a second example, consider the following simple task: Two parties (say, two databases), each having a list of items, wish to find out which items appear in both lists. This task is somewhat more complex than the previous one, since both parties have private inputs and both have private outputs which are different than, but related to, each other. Still, it can be formulated as a function in a straightforward way: $f_{\text{DI}}((x^1_1,...,x^1_n),(x^2_1,...,x^2_m)) = ((b^1_1,...,b^1_n),(b^2_1,...,b^2_m))$, where $b^i_j = 1$ if x^i_j equals $x^{3-i}_{j'}$ for some j', and $b^i_j = 0$ otherwise. This would mean that a party P which follows the protocol is guaranteed to get a valid answer based on its own database x and *some* database x', where x' was determined by the other party *based only on the initial input of the other party.* Furthermore, the information learned by the other party is computed based on the same two values x and x'. Also, if there is reason to believe that the other party used some concrete "real" database x', then correctness is guaranteed with respect to that specific x'. Recall, however, that the definition does not guarantee fairness. That is, the other party may obtain the output value first, and based on that value decide whether P will obtain its output value. In Section 4 we will see how to express fairness within an extended formalism.

Common Randomness. Finally, we consider a task that involves distributional requirements from the outputs of the parties. Specifically, we consider the task of generating a common string that is guaranteed to be taken from a pre-defined distribution, say the uniform distribution over the strings of some length: $f^k_{\text{CR}}(-,-) = (r,r)$, where r is a random k-bit string. Here the parties are guaranteed that the output r is distributed (pseudo)randomly over $\{0,1\}^k$. Furthermore, each party is guaranteed that the other party does not have any "trapdoor information" on r that cannot be efficiently computed from r alone. As mentioned in the Introduction, this guarantee becomes crucial in some cryptographic applications. Finally, as in the previous case, fairness is not guaranteed.

4 Basic security: The general case

Section 3 provides a framework for defining security of a restricted class of protocols for a restricted class of tasks: protocols that involve only two parties, and tasks that can be captured as two-party functions. While this case captures much of the essence of the general notion, it leaves much to be desired in terms of the expressibility and generality of the definitional paradigm.

This section generalizes the treatment of Section 3 in several ways, so as to capture a wider class of cryptographic tasks. First we consider *multi-party* tasks, namely tasks where multiple (even unboundedly many) parties contribute inputs and obtain outputs. This requires expressing various synchrony and scheduling concerns. Next, we consider also tasks which require security against "the network", namely against parties that do not take legitimate part in the protocol but may have access to the communication. Third, we consider also "reactive tasks," where a party provides inputs and obtains outputs multiple times, and new inputs may depend on previously obtained outputs. Fourth, we allow expressing situations where parties get "corrupted", or "broken into" in an adaptive way throughout the computation. Finally, we provide the ability to formulate weaker requirements, which allows protocols where the legitimate outputs do not constitute any pre-defined function of the inputs, and can be potentially influenced by the adversary in some limited way. While we describe all the extensions together, many are independent of each other and could be considered separately. We try to identify the effect of each individual extension as we go along.

Still, throughout this section we only consider the case of a single execution of a protocol, run in isolation. Treatment of systems where multiple protocol executions co-exist is deferred to the next sections.

The necessary extensions to the basic system model are presented first, in Section 4.1. Section 4.2 presents the extensions to the definition of security, while Section 4.3 provides some additional examples. Finally, Section 4.4 briefly reviews some basic feasibility results for this definition.

4.1 The system model

In many respects, the system model from Section 3.1 suffices for capturing general multi-party protocols and their security. (In fact, some existing formalisms offer comparable generality, in the sense that they do not include the extensions described below.) Still, that model has some limitations: First, it can only handle a *fixed number* of interacting ITIs. This suffices for protocols where the number of participants is fixed. However, it does not allow modeling protocols where the number of parties can grow in an adaptive way based on the execution of the protocol, or even only as a function of a security parameter. Such situations may indeed occur in real life, say in an on-line auction or gambling application. Another limitation is that the addressing mechanism for external write requests is highly idealized, and does not allow for natural modeling of routing and identity management issues. While this level of abstraction is sufficient for systems with small number of participants that know each other in advance, it does not suffice for open systems, where parties may learn about each other only via the protocol execution.

We thus extend the model of Section 3.1 in two ways. First, we allow for new ITIs to be added to the system during the course of the computation. This is done via a special "invoke new ITI" instruction that can be executed by a currently running ITI. The code of the new ITI should be specified in the invocation instruction. The effect of the instruction is that a new ITI with the specified code is added to the system. The externally writable tapes of the new ITI can now be written to by other ITIs. Note that, given the new formalism, a system of ITMs can now be specified by a single ITM, the initial ITM, along with the control function. All other ITIs in the system can be generated dynamically during the course of the execution. The notion of PT ITMs from Section 3.1 remains valid, in the sense that it is still guaranteed that a system of ITMs is guaranteed to complete each execution in polynomial time, as long as the initial ITM is PT and the control function is polynomially computable.

The second change is to add a special identity tape to the description of an ITM. This tape will be written to once, upon invocation, and will be readable by the ITM itself. This means that the behavior of the ITM can depend on its identity (namely on the contents of its identity tape). Furthermore, an external write instruction will now specify the target ITM via its identity, rather than via a "dedicated link" (represented via some external index).

The identity of an ITI is determined by the ITI that invokes it. To guarantee unambiguous addressing, we require that identities (often dubbed IDs) be unique. That is, an invocation instruction that specifies an existing ID is rejected. (This rule can be implemented, say, by the control function.)

4.2 Definition of Security

We extend the definition of security in several steps. First, we extend the model of protocol execution. Next, we extend the ideal process. Finally, we extend the notion of realizing a trusted party. As we'll see, it often turns out that taking a more general view actually simplifies some aspects of the formalism.

The protocol execution experiment. We describe the generalized protocol execution experiment. Let π be a protocol to be executed. (π need not specify the number of participants in advance.) As before, the model for executing a protocol π is parameterized by two additional ITMs, an environment $\mathcal{E}$ and an adversary $\mathcal{A}$.

The environment is the initial ITI. It first invokes the adversary $\mathcal{A}$, and from then on can invoke as many ITI as it wishes, as long as they all run the code of π. In particular, $\mathcal{E}$ can determine the identities of these ITIs, or parties. In addition, $\mathcal{E}$ can write to the input tapes of the parties throughout the computation. However, it cannot send further information to $\mathcal{A}$ after its invocation. Parties can write to the subroutine output tapes of $\mathcal{E}$. No other interaction between $\mathcal{E}$ and the system is allowed.

Once a party is activated, either with an input value, or with an incoming message (i.e., a value written on the incoming communication tape), it follows its code and potentially generates an outgoing message or an output. All outgoing messages are handed to the adversary (i.e., written to its incoming communication tape), regardless of the stated destinations of the messages. Outputs are written on the subroutine output tape of $\mathcal{E}$. Parties may also invoke new ITIs, that may run either π or another code. However, these ITIs are not allowed to directly communicate with $\mathcal{E}$ (i.e., they cannot write to the tapes of $\mathcal{E}$).

Once the adversary is activated, it can deliver a message to a party, i.e. write the message on the party's incoming communication tape. In its last activation it can also generate an output, i.e. write the output value on the incoming communication tape of $\mathcal{E}$.

As before, the final output of the execution is the (one bit) output of the environment. With little chance of confusion, we re-define the notation $\text{EXEC}_{\pi,\mathcal{A},\mathcal{E}}$ to refer to the present modeling.

The ideal process. The main difference from the ideal process in Section 3 is that, instead of considering only trusted parties that perform a restricted set of operations (such as evaluating a function), we let the trusted party run arbitrary code, and in particular to repeatedly interact with the parties, as well as directly with the adversary. In addition, the richer system model allows us to simplify the presentation somewhat by formulating the ideal process as a special case of the general protocol execution experiment.

That is, the security requirements of a cryptographic task should first be formalized in terms of code for a trusted party, called an ideal functionality (some examples appear below). Given an ideal $\mathcal{F}$, we define an ideal protocol $I_{\mathcal{F}}$ as follows: When a party running $I_{\mathcal{F}}$ obtains an input value, it immediately copies this value to the input of $\mathcal{F}$. (The first party to do so will also invoke $\mathcal{F}$.) When a party receives an output from

$\mathcal{F}$ (on its subroutine output tape), it immediately outputs this value (i.e., copies it to the subroutine output tape of $\mathcal{E}$). All other activations are ignored.

The notation $\text{IDEAL}_{\mathcal{F},\mathcal{A},\mathcal{E}}$ from Section 3.2 is no longer needed; it is replaced by $\text{EXEC}_{I_{\mathcal{F}},\mathcal{A},\mathcal{E}}$.

Protocol emulation and secure realization. The notion of realizing an ideal process remains essentially the same. Yet, formalizing the ideal process as an execution of a special type of a protocol allows formalizing the definition of realizing an ideal functionality as a special case of the more general notion of emulating one protocol by another. That is:

Definition 2 (Protocol emulation with basic security) *A protocol π* emulates *protocol ϕ if for any PT adversary $\mathcal{A}$ there exists a PT adversary $\mathcal{S}$ such that for all PT environments $\mathcal{E}$ that output only one bit:*

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{E}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \tag{2}$$

Definition 3 (Realizing functionalities with basic security) *A protocol π* realizes *an ideal functionality $\mathcal{F}$ if π emulates $I_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$.*

4.2.1 Discussion

Some modeling decisions. We highlight some characteristics of the extended model of protocol execution. First, the present model continues to model the environment and adversary as *centralized* entities that have global views of the distributed computation. While in the two-party case this was a natural choice, in the multi-party case this modeling becomes an abstraction of reality. This modeling seems instrumental for capturing security in an appropriate way, since we would want security to hold *even when* the adversarial entities do have global view of the computation. Still, it introduces some "slack" to the definition of security, in that it allows the simulator to be centralized even when the adversarial activity is not.

Another point is the restricted communication between $\mathcal{E}$ and $\mathcal{A}$. Recall that $\mathcal{E}$ cannot directly provide information to $\mathcal{A}$ other than at invocation time, and $\mathcal{A}$ can directly provide information to $\mathcal{E}$ only at the end of its execution. (Of course, $\mathcal{E}$ and $\mathcal{A}$ can exchange information indirectly, via the parties, but this type of exchange is limited by the properties of the specific protocol π in question.) This restriction is indeed natural in a stand-alone setting, since there is no reason to let the adversarial activity against the protocol depend in an artificial way on the local inputs and outputs of the non-corrupted parties. Furthermore, it is very important technically, since it allows proving security of protocols that are intuitively secure. See more discussion in Section 3.3.

Also, note that the present modeling of asynchronous scheduling of events, while typical in cryptography, is different than the standard modeling of asynchronous scheduling in general distributed systems, such as those mentioned in Section 3.1. In particular, there asynchrony is typically captured via *non-deterministic* scheduling, where the non-determinism is resolved by an all-powerful scheduler that has access to the entire current state of the system. Here, in contrast, the scheduling is determined by the environment and adversary, namely in an algorithmic and computationally bounded way. This modeling of asynchrony, while admittedly weaker, seems essential for capturing security that holds only against computationally bounded attacks. Combining non-deterministic and adversarial scheduling is an interesting challenge.

Modeling various corruption and communication methods. The simplified model of Section 3 essentially postulates that one of the two parties is "corrupted," that is it runs arbitrary code irrespective of the protocol instructions. Furthermore, and this party is corrupted in advance, before the protocol starts. In contrast, the extended model postulates that all parties follow the specified protocol π; no deviations are

allowed. Instead, deviations from the original protocol are captured as additional protocol instructions that "get activated" upon receiving special corruption messages from the adversary. For instance, to capture arbitrary deviation from the protocol, instruct a party to follow the adversary's instructions once it receives a corruption messages. To capture parties that continue following the protocol but pool all their information together (aka honest-but-curious corruptions, a party that receives a corruption message will send all its internal state to the adversary, and otherwise continue to follow the protocol. Other types of corruptions can be captured analogously. This way of modeling corruptions has two advantages: first it simplifies the description of the basic model, and second it provides flexibility in considering multiple types of corruptions within the same model, and even within the same execution.

The above experiment gives the adversary full control over the communication, thus representing completely asynchronous, unreliable and unauthenticated communication. More abstract communication models, providing various levels of authentication, secrecy, reliability and synchrony, can be captured by appropriately restricting the adversary. (For instance, to model authenticated communication, restrict the adversary to deliver only messages that were previously sent by parties, and include the identity of the source within each message.) In addition, as will be seen in subsequent sections, all these communication models can be captured as different abstractions within the same basic model, rather than having to re-define the underlying model for each one.

On the generalized modeling of the ideal process. Modeling the trusted party as a general ITM greatly enhances the expressibility of the definitional framework, in terms of the types of concerns and levels of security that can be captured. Indeed, it becomes possible to "fine-tune" the requirements at wish. The down side of this generality is that the exact security implication of a given ideal functionality (or, "code for the trusted party") is not always immediately obvious, and small changes in the formulation often result in substantial changes in the security requirements. Here we very briefly try to highlight some salient aspects of the formalism, as well as useful "programming techniques" for ideal functionalities.

Two obvious aspects of the general formulation are that it is now possible to formulate *multi-party* and *reactive* tasks. In addition, letting the ideal functionality interact directly with the adversary in the ideal process (namely, with the "simulator") has two main effects. First, providing information to the adversary can be naturally used to capture the "allowed leakage of information" by protocols that realize the task. For instance, if some partial information on the output value can be leaked without violating the requirements, then the ideal functionality might explicitly hand this partial information to the adversary.

Receiving information directly from the adversary is useful in capturing the "allowed influence" of the adversary on the computation. For instance, if the timing of a certain output event is allowed to be adversarially controlled (say, within some limits), then the ideal functionality might wait for a trigger from the adversary before generating that output. Alternatively, if several different output values are legitimate for a given set of inputs, the ideal functionality might let the adversary choose the actual output within the given constraints. In some cases it might even be useful to let the adversary hand some arbitrary *code* to be executed by the ideal functionality in a "monitored way," namely subject to constraints set by the ideal functionality.

In either case, since the direct communication between the ideal functionality and the adversary is not part of the input-output interface of the actual parties, the effect of this communication is always only to *relax* the security requirements of the task.

An example of the use of direct communication between the adversary and the ideal functionality is the modeling of the allowable information leakage and adversarial influence upon party corruption. In the ideal process, party corruption is captured via a special message from the adversary to the ideal functionality. In response to that message, the ideal functionality might provide the adversary with appropriate information

(such as past inputs and outputs of the corrupted party), allow the adversary to change the contributed input values of the corrupted parties, or even change its behavior in more global ways (say, when the number of corrupted parties exceeds some threshold).

Finally, recall that the ideal functionality receives input directly from the environment, and provides outputs directly to the environment, without intervention of the adversary. This has the effect that an ideal protocol can guarantee delivery of messages, as well as concerns like *fairness,* in the sense that one party obtains output if and only if another party does. In fact, special care should be taken, when writing an ideal functionality, to make sure that the functionality allows the adversary to delay delivery of outputs (say, by waiting for a trigger from the adversary before actually writing to the subroutine output tape of the recipient party); otherwise the specification may be too strong and unrealizable by a distributed protocol.

4.3 More examples

Definition 3 allows capturing the security and correctness requirements of practically any distributed task, in a stand-alone setting. This includes, e.g., all the tasks mentioned in the introduction. Here we sketch ideal functionalities that capture the security requirements of two basic tasks. Each example is intended to highlight different aspects of the formalism.

As stipulated in the system model, each input to an ideal functionality has to include the identity. For brevity, we omit these identities from the description. Indeed, for security in a stand-alone setting it is not essential that the functionality's identity will be available to the protocol. In contrast, for security in a multi-instance system, considered in subsequent sections, making the identity available to the protocol is often essential for meaningful realizations.

Commitment. First we formulate an ideal functionality that captures the security requirements from a commitment protocol, as informally sketched in the introduction. Commitment is inherently a *two step process*, namely commitment and opening. Thus it cannot be naturally captured within the formalism of Section 3, in spite of the fact that it is a two-party functionality.

The ideal commitment functionality, $\mathcal{F}_{\text{COM}}$, formalizes the "sealed envelope" intuition in a straightforward way. That is, when receiving from the committer C an input requesting to commit to value x to a receiver R, $\mathcal{F}_{\text{COM}}$ records (x, R) and notifies R and the adversary that C has committed to some value. (Notifying the adversary means that the fact that a commitment took place need not be hidden.) The opening phase is initiated by the committer inputting a request to open the recorded value. In response, $\mathcal{F}_{\text{COM}}$ outputs x to R and the adversary. (Giving x to the adversary means that the opened value can be publicly available.)

In order to correctly handle adaptive corruption of the committer during the course of the execution, $\mathcal{F}_{\text{COM}}$ responds to a request by the adversary to corrupt C by first outputting a corruption output to C, and then revealing the recorded value x to the adversary. In addition, if the `Receipt` value was not yet delivered to R, then $\mathcal{F}_{\text{COM}}$ allows the adversary to modify the committed value. This last stipulation captures the fact that the committed value is fixed only at the end of the commit phase, thus if the committer is corrupted during that phase then the adversary might still be able to modify the committed value.

$\mathcal{F}_{\text{COM}}$ is described in Figure 1. For brevity, we use the following terminology: The instruction "send a delayed output x to party P" should be interpreted as "send (x, P) to the adversary; when receiving `ok` from the adversary, output x to P."

Realizing $\mathcal{F}_{\text{COM}}$ is a stronger requirement than the basic notions of commitment in the literature (see e.g. [G01]). In particular, this notion requires both "extractability" and "equivocality" for the committed value. These notions (which are left undefined here) become important when using commitment within other protocols; they are discussed in subsequent sections, as well as in [CF01, C01]. Still, $\mathcal{F}_{\text{COM}}$ is realizable by

Functionality $\mathcal{F}_{\text{COM}}$

1. Upon receiving an input $(\texttt{Commit}, x)$ from party C, record (C, R, x) and generate a delayed output $(\texttt{Receipt})$ to R. Ignore any subsequent $(\texttt{Commit}...)$ inputs.
2. Upon receiving an input $(\texttt{Open})$ from C, do: If there is a recorded value x then generate a delayed output $(\texttt{Open}, x)$ to R. Otherwise, do nothing.
3. Upon receiving a message $(\texttt{Corrupt}, C)$ from the adversary, output a $\texttt{Corrupted}$ value to C, and send x to the adversary. Furthermore, if the adversary now provides a value x', and the $(\texttt{Receipt})$ output was not yet written on R's tape, then change the recorded value to x'.

Figure 1: The Ideal Commitment functionality, $\mathcal{F}_{\text{COM}}$

standard constructions, assuming authenticated communication channels.

Key Exchange. Key exchange (KE) is a task where two parties wish to agree on a random value (a "key") that will remain secret from third parties. Typically, the key is then used to encrypt and authenticate the communication between the two parties. Key exchange may seem reminiscent of the coin-tossing task, discussed in Section 3.3. However, it is actually quite different: Essentially, in the case of key exchange the two parties wish to jointly thwart an external attacker, whereas in coin-tossing the parties wish to protect themselves from *each other.* More precisely, for key exchange we only care about the fact that the key is random when both parties follow their protocol. On the other hand, for coin-tossing the agreed value need not be kept secret from third parties (embodied by the adversary). Furthermore, since key exchange is usually carried out in a multi-party environment with asynchronous and unauthenticated communication, issues such as precise timing of events and binding of the output key to specific identities become crucial. Thus, modeling of key exchange naturally involves an interactive interface, as well communicating directly with the adversary.

Functionality $\mathcal{F}_{\text{KE}}$, presented in Figure 2, proceeds as follows. Upon receiving an $(\texttt{Initiate}, I, R)$ input from some party I (called the *initiator*), $\mathcal{F}_{\text{KE}}$ sends a delayed output $(\texttt{Initiate}, I)$ to R. Upon receiving the input $(\texttt{Respond})$ from R, $\mathcal{F}_{\text{KE}}$ forwards this input to the adversary. Now, when receiving a value $(\texttt{Key}, P, \tilde{k})$ from the adversary, $\mathcal{F}_{\text{KE}}$ first verifies that $P \in \{I, R\}$, else P gets no output. If the two peers are currently uncorrupted, then P obtains a truly random and secret key κ for that session. If any of the peers is corrupted then P receives the key $\tilde{k}$ determined by the adversary.

Functionality $\mathcal{F}_{\text{KE}}$

1. Upon receiving an input $(\texttt{Initiate}, I, R)$ from party I, send a delayed output $(\texttt{Initiate}, I)$ to R. Upon receiving $(\texttt{Respond})$ from party R, send $(\texttt{Respond})$ to the adversary.
2. Upon receiving a message $(\texttt{Corrupt}, P)$ from the adversary, for $P \in \{I, R\}$, mark P as corrupted and output $(\texttt{Corrupted})$ to P.
3. Upon receiving a message $(\texttt{Key}, P, \tilde{k})$ from the adversary, for $P \in \{I, R\}$ do:
   (a) If there is no recorded key κ then choose $\kappa \xleftarrow{R} \{0,1\}^k$ and record κ.
   (b) If neither I nor R are corrupted then output $(\texttt{Key}, \kappa)$ to P. Else, output $(\texttt{Key}, \tilde{k})$ to P.

Figure 2: The Key Exchange functionality, $\mathcal{F}_{\text{KE}}$

$\mathcal{F}_{\text{KE}}$ Attempts to make only a minimal set of requirements from a candidate protocol. In particular, it attempts o allow the adversary maximum flexibility in determining the order in which the parties obtain their outputs. Also, the fact that there is no requirement on the key when one of the parties is corrupted is captured by allowing the adversary to determine the key in this case. Still, $\mathcal{F}_{\text{KE}}$ guarantees that if two uncorrupted parties locally obtain a key, then they obtain the same value, and this value is uniformly generated and independent from the adversary's view.

Key Exchange is impossible to realize without some form of authentication set-up, say pre-shared keys, authentication servers, or public-key infrastructure. Still, the formulation of $\mathcal{F}_{\text{KE}}$ is agnostic to the particular set-up in use. It only specifies the desired overall functionality. In each of these cases, $\mathcal{F}_{\text{KE}}$ is realizable by standard protocols, both with respect to basic security and with respect to UC security, discussed in the second part of this tutorial.

Byzantine Agreement. Next we formulate an ideal functionality that captures (one variant of) the Byzantine Agreement task. Here each party has binary input, and the parties wish to output a common value with the only restriction that if all parties have the same input value then they output that value. The functionality, $\mathcal{F}_{\text{BA}}$, is presented in Figure 3. Let us highlight some aspects of its formulation. First, the number of parties (which is a parameter to $\mathcal{F}_{\text{BA}}$) can depend on the environment. Also the identities of the participants can be determined adaptively as they join the protocol. Second, the fact that the adversary is notified on any new input captures the fact that privacy of the inputs of the parties is not guaranteed. Third, $\mathcal{F}_{\text{BA}}$ allows the output value to take any adversarially chosen value, unless all parties have the same input. (In particular, the parties are not guaranteed to compute any pre-determined function of their inputs.) Four, $\mathcal{F}_{\text{BA}}$ captures a *blocking* primitive, namely no party obtains output unless all parties provide inputs. It also guarantees *fair* output delivery: As soon as one party obtains its output, all parties who ask for their output receive it without delay. (Note that if $\mathcal{F}_{\text{BA}}$ would have simply sent the outputs to all parties, then fairness would not have been guaranteed since the adversary could have prevented the delivery to some parties by not returning control to $\mathcal{F}_{\text{BA}}$.) Finally, $\mathcal{F}_{\text{BA}}$ does not have a postulation for the case of party corruption. This captures the fact that corrupting a party should give no advantage to the adversary.

Functionality $\mathcal{F}_{\text{BA}}$

$\mathcal{F}_{\text{BA}}$ proceeds as follows, when parameterized by the number n of participants. A set $\mathcal{P}$ of participant identities is initialized to empty. Then:

1. Upon receiving input `(Input,`v`)` from some new party $P \notin \mathcal{P}$, where $v \in \{0,1\}$, add P to $\mathcal{P}$, set $x_P = v$, and send a message `(Input,`P, v`)` to the adversary. As soon as $|\mathcal{P}| = n$, ignore additional `(Input...)` inputs.
2. Upon receiving input `(Output)` from a party $P \in \mathcal{P}$, if $|\mathcal{P}| < n$ then do nothing. Else:
   (a) If the output value y is not yet determined then determine y as follows: If there exists a value $b \in \{0,1\}$ such that $x_P = b$ for all parties $P \in \mathcal{P}$, then set $y = b$. Else, obtain a value y from the adversary.
   (b) Output y to P.

Figure 3: The Byzantine Agreement functionality, $\mathcal{F}_{\text{BA}}$

Note that $\mathcal{F}_{\text{BA}}$ is agnostic to the specific model of computation in which it is realized. Naturally, realizing $\mathcal{F}_{\text{BA}}$ requires different techniques in different settings (depending e.g. on the level of synchrony and the specific authentication set-up). We conjecture that, in each such setting, realizing $\mathcal{F}_{\text{BA}}$ is essentially

equivalent to the standard definition of the primitive in that model. (In particular, it is easy to see that if half or more of the parties are corrupted then $\mathcal{F}_{\text{BA}}$ becomes unrealizable in *any* computational model. Indeed, in such settings the *Byzantine Broadcast* formulation, where only one party has input, is preferable.)

4.4 Feasibility

We very briefly mention some of the basic feasibility results for cryptographic protocols, which establish a remarkable fact: Practically any cryptographic task can be realized, in principle, by a polynomial-time interactive protocol.

The first work to provide a general feasibility result is Yao [Y86], which showed how to securely evaluate any two-party function by a two-party protocol, in a setting which corresponds to that of Section 3, in the case of "honest-but-curious corruptions" where even corrupted parties continue to follow the protocol.

The basic idea is as follows. Given a function f, first have one party, X, with input x, prepare a binary circuit C_x^f such that for any y, $C_x^f(y) = f(x, y)$. Then X sends to the other party, Y, an "obfuscated version" of C_x^f, so that Y can only evaluate C_x^f on a single input of its choice, without learning any additional information on the "internals" of C_x^f. The obfuscation method involves preparing a "garbled version" of each gate in the circuit, plus allowing Y to obtain a matching "garbled version" of one of the possible two values of each input line. Given this information, Y will be able to evaluate the circuit in a gate by gate fashion, and obtain a "garbled version" of the output line of the circuit. Finally, X will send Y a table that maps each possible garbled value of the output line to the corresponding real value.

Goldreich, Micali and Wigderson [GMW87] generalize [Y86] in two main respects. First, they generalize Yao's "obfuscated circuit" technique to *multi-party* functions. Here all parties participate in evaluating the "garbled gates". Further generalization to reactive functionalities can be done in a straightforward way, as demonstrated in [CLOS02].

Perhaps more importantly, [GMW87] generalize Yao's paradigm to handle also *Byzantine* corruptions, where corrupted parties may deviate from the protocol in arbitrary ways. This is done via a generic and powerful application of Zero-Knowledge protocols. A somewhat over-simplified description of the idea follows: In order to obtain a protocol π that realizes some task for Byzantine corruptions, first design a protocol π' that realizes the task for honest-but-curious corruptions. Now, in protocol π each party P runs the code of π', and in addition, along with each message m sent by π', P sends a Zero-Knowledge proof that the message m was computed correctly, according to π', based on *some* secret input and the (publicly available) messages that P received. The protocols of [GMW87] withstand any number of faults, without providing *fairness* in output generation. Fairness is guaranteed only if the corrupted parties are a minority.

Ben-Or, Goldwasser and Wigderson [BGW88] demonstrate, using algebraic techniques, that if the parties are equipped with ideally secret pairwise communication channels, then it is possible to securely evaluate any multi-party function in a *perfect way* (see discussion following Definition 1), in the presence of honest-but-curious corruption of any minority of the parties. The same holds even for Byzantine corruptions, as long as less only less than a *third* of the parties are corrupted. Rabin and Ben Or [RB89] later improved the bound from a third to any minority, assuming a broadcast channel. These bounds are tight. A nice feature of the [BGW88] protocols is that, in contrast to the [GMW87] protocols, they are secure even against adaptive corruptions.

All the above results assume ideally authenticated communication. If an authenticated set-up stage is allowed, then obtaining authenticated communication is simple, say by digitally signing each message relative to pre-distributed verification keys. When no authenticated set-up is available, however, then no task that requires some form of authentication of the participants can be realized. Still, as shown in Barak et.al. [B+05], an "unauthenticated variant" of any cryptographic task can still be realized, much in the

spirit of [Y86, GMW87], even without any authenticated set-up. Interestingly, the proof of this result uses in an essential way protocols that are *securely composable,* namely retain their security properties even when running together in the same system. This can be seen as a demonstration of the fact that secure composability, discussed next, is in fact a very basic security requirement for cryptographic protocols.

Acknowledgments. My thinking and understanding of cryptographic protocols has been shaped over the years by discussions with many insightful researchers, too numerous to mention here. I thank you all. Oded Goldreich and Hugo Krawczyk were particularly influential, with often conflicting (complementary?) views of the field. I'm also grateful to the editor, Sergio Rajsbaum, for his effective blend of flexibility and persistence.

References

[BPW04] M. Backes, B. Pfitzmann, and M. Waidner. Secure Asynchronous Reactive Systems. Eprint archive, http://eprint.iacr.org/2004/082, March 2004.

[B+05] B. Barak, R. Canetti, Y. Lindell, R. Pass and T. Rabin. Secure Computation Without Authentication. In *Crypto'05*, 2005.

[B91] D. Beaver. Secure Multi-party Protocols and Zero-Knowledge Proof Systems Tolerating a Faulty Minority. J. Cryptology, (1991) 4: 75-122.

[BCG93] M. Ben-Or, R. Canetti and O. Goldreich. Asynchronous Secure Computations. *25th Symposium on Theory of Computing (STOC)*, ACM, 1993, pp. 52-61.

[BGW88] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *20th Symposium on Theory of Computing (STOC)*, ACM, 1988, pp. 1-10.

[B82] M. Blum. Coin flipping by telephone. IEEE Spring COMPCOM, pp. 133-137, Feb. 1982.

[BCC88] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *JCSS*, Vol. 37, No. 2, pages 156–189, 1988.

[C00] R. Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, Vol. 13, No. 1, winter 2000.

[C01] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. (Earlier version of the present work.) Available at http://eccc.uni-trier.de/eccc-reports/2001/TR01-016/revision01.ps. Extended abstract in *42nd FOCS*, 2001.

[C+06] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala. Task-Structured Probabilistic I/O Automata. In *Workshop on discrete event systems (WODES)*, 2006.

[CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. Crypto '01, 2001.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, A. Sahai. Universally composable two-party and multi-party secure computation. *34th STOC*, pp. 494–503, 2002.

[CGKS95] B. Chor, O. Goldreich, E. Kushilevitz, M. Sudan. Private Information Retrieval. *36th FOCS*, 1995, pp. 41-50.

[DM00] Y. Dodis and S. Micali. Secure Computation. *CRYPTO '00*, 2000.

[G01] O. Goldreich. *Foundations of Cryptography.* Cambridge Press, Vol 1 (2001) and Vol 2 (2004).
NP.

[GMW87] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game. *19th Symposium on Theory of Computing (STOC)*, ACM, 1987, pp. 218-229.

[GL90] S. Goldwasser, and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. *CRYPTO '90, LNCS 537*, 1990.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *JCSS*, Vol. 28, No 2, April 1984, pp. 270-299.

[GMRa89] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Comput.*, Vol. 18, No. 1, 1989, pp. 186-208.

[HM00] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. *Journal of Cryptology*, Vol 13, No. 1, 2000, pp. 31-60. Preliminary version in *16th Symp. on Principles of Distributed Computing (PODC)*, ACM, 1997, pp. 25–34.

[H85] C. A. R. Hoare. Communicating Sequential Processes. International Series in Computer Science, Prentice Hall, 1985.

[LMMS98] P. Lincoln, J. Mitchell, M. Mitchell, A. Scedrov. A Probabilistic Poly-time Framework for Protocol Analysis. *5th ACM Conf. on Computer and Communication Security*, 1998, pp. 112-121.

[Ly96] N. Lynch. *Distributed Algorithms.* Morgan Kaufman, San Francisco, 1996.

[LSV03] N. Lynch, R. Segala and F. Vaandrager. Compositionality for Probabilistic Automata. *14th CONCUR*, LNCS vol. 2761, pages 208-221, 2003. Fuller version appears in MIT Technical Report MIT-LCS-TR-907.

[MMS03] P. Mateus, J. C. Mitchell and A. Scedrov. Composition of Cryptographic Protocols in a Probabilistic Polynomial-Time Process Calculus. *CONCUR*, pp. 323-345. 2003.

[MR91] S. Micali and P. Rogaway. Secure Computation. unpublished manuscript, 1992. Preliminary version in *CRYPTO '91, LNCS 576*, 1991.

[M89] R. Milner. Communication and Concurrency. Prentice Hall, 1989.

[M99] R. Milner. Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, 1999.

[MRST01] J. Mitchell, A. Ramanathan, A. Scedrov, V. Teague. A probabilistic polynomial-time calculus for analysis of cryptographic protocols (Preliminary report). 17-th Annual Conference on the Mathematical Foundations of Programming Semantics, Arhus, Denmark, May, 2001, *ENTCS Vol. 45 (2001).*

[PW94] B. Pfitzmann and M. Waidner. A general framework for formal notions of secure systems. Hildesheimer Informatik-Berichte 11/94, Universitat Hildesheim, 1994. Available at http://www.semper.org/sirene/lit.

[PW00] B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. *7th ACM Conf. on Computer and Communication Security*, 2000, pp. 245-254.

[PW01] B. Pfitzmann and M. Waidner. A model for asynchronous reactive systems and its application to secure message transmission. IEEE Symposium on Security and Privacy, May 2001. Preliminary version in http://eprint.iacr.org/2000/066 and IBM Research Report RZ 3304 (#93350), IBM Research, Zurich, December 2000.

[RB89] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. *21st Symposium on Theory of Computing (STOC)*, ACM, 1989, pp. 73-85.

[Y82A] A. Yao. Protocols for Secure Computation. In *Proc. 23rd Annual Symp. on Foundations of Computer Science (FOCS)*, pages 160–164. IEEE, 1982.

[Y86] A. Yao, How to generate and exchange secrets, In *Proc. 27th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.