Adaptive Security for Threshold Cryptosystems

Extended version of a paper that appeared under the same title in Crypto'99

Ran Canetti[†], Rosario Gennaro[†], Stanisław Jarecki[*], Hugo Krawczyk[‡] and Tal Rabin[†]

August 31, 1999

Abstract

We present *adaptively-secure* efficient solutions to several central problems in the area of threshold cryptography. We prove these solutions to withstand *adaptive attackers* that choose parties for corruption at any time during the run of the protocol. In contrast, all previously known efficient protocols for these problems were proven secure only against less realistic *static adversaries* that choose and fix the subset of corrupted parties before the start of the protocol run.

Specifically, we provide adaptively-secure solutions for distributed key generation in discrete-log based cryptosystems, and for the problem of distributed generation of DSS signatures (threshold DSS). We also show how to transform existent static solutions for threshold RSA and proactive schemes to withstand the stronger adaptive attackers. In doing so, we introduce several techniques for the design and analysis of adaptively-secure protocols that may well find further applications.

Key Words: Threshold Cryptography, Adaptive Security, Proactive Security, Zero-knowledge Proofs

1 Introduction

Distributed cryptography has received a lot of attention in modern cryptographic research. It covers a variety of areas and applications, from the study of secret-sharing schemes, to the distributed computation of general functions using secure multi-party protocols, to the design and analysis of specific threshold cryptosystems. Two main goals that motivate this research area are: (i) provide security to applications that are inherently distributed, namely, several parties are trying to accomplish some common task (e.g., secure elections, auctions, games) in the presence of an attacker, and (ii) avoid single points-of-failure in a security system by distributing the crucial security resources (e.g. sharing the ability to generate signatures). In both cases the underlying assumption is that an attacker can penetrate and control a portion of the parties in the system but not all of them.

Coming up with correct protocols for meeting the above goals has proven to be a challenging task; trying to design protocols that are practical as well as fully-analyzed is even more challenging. One inherent difficulty in the analysis of cryptographic systems, in general, is the need to define a mathematical model that is strong enough to capture realistic security threats and attacker's capabilities, and which, at the same time, allows to prove the security of sound solutions. Due to the complexity of building and reasoning about distributed protocols, this difficulty is even greater in the area of distributed cryptography. In this case a large variety of security models have been proposed. This is not just due to "philosophical disagreements" on what the best way to model reasonable attackers is, but it is also influenced by our ability (or lack of it) to create protocols for which we can prove security under strong adversarial models.

One major distinction between security models for distributed protocols is whether the attacker is *static* or *adaptive* (the latter is also called "dynamic"). In both cases the attacker is allowed to corrupt any subset of parties up to some size (or threshold) as specified by the security model. However, in the case of an adaptive adversary the decision of which parties to corrupt can be made at any time during the run of the protocol and, in particular, it can be based on the information gathered by the attacker during this run. In contrast, in the static case the attacker must choose its victims independently of the information it learns during the protocol. Therefore, the subset of corrupted parties can be seen as chosen and fixed by the attacker before the start of the protocol's run.

The bulk of published works on distributed cryptography assumes a static adversary. This is due to the difficulties encountered when trying to design and prove protocols resistant to adaptive adversaries. However, it is known that protocols can be secure in the static model and still insecure in the adaptive one [CFGN96, CDD$^+$99, Can98]. Therefore, since the adaptive adversary model appears to better capture real threats, proving protocols secure in the static model is not enough to assume their practical security. Although general constructions have been shown in the adaptive-adversary model for secure distributed evaluation of any polynomial-time computable function [BGW88, CCD88, BH92, CFGN96], these general results do not provide sufficiently efficient solutions to practical applications like adaptively-secure threshold DSS or RSA signature scheme. Until now, no efficient adaptive solutions for threshold cryptosystems were known.

Our Contribution. The main contribution of this paper is in providing concrete, fully-specified, fully-analyzed solutions to some of the central problems in threshold cryptography, and proving their security in the adaptive adversary model. Our solutions add little overhead relative to existing solutions for the same problems in the static-adversary model. They are also constant-round; namely, the number of rounds of communication is fixed and independent of the number of parties in the system, the input length, or the security parameter. Thus we believe that our protocols are practical and can be of real use in emerging threshold cryptography applications. Very importantly, we provide full analysis and proofs for our solutions. This is essential in an area where simple intuition is usually highly misleading and likely to produce flawed protocols.

Our work introduces general new techniques for the design and analysis of protocols in the adaptive-adversary model. In section 2 we give an overview of these techniques to aid the understanding of the protocols and the proofs given in the following sections. We also hope that these techniques will be applicable to the design of adaptively-secure solutions to other problems.

In section 3 we start the presentation of our protocols with an adaptively-secure solution for the distributed generation of keys for DSS, El Gamal, and other discrete-log based public-key systems (for signatures and encryption). Previous solutions to this problem worked against static adversaries only. Such a protocol is not only needed for generating keys in a distributed way without a trusted party, but it is also a crucial component of many other cryptographic protocols. We illustrate this in our own work by using it for distributed generation of the $r = g^k$ part in DSS signatures.

In section 4 we present a threshold DSS scheme consisting of the above distributed key generation and an adaptively-secure protocol for the shared generation of DSS signatures. We first show a simplified protocol where the attacker can control up to $t < n/4$ players. This protocols helps in highlighting and understanding many of our new and basic techniques. Next we describe an optimal-resilience, $t < n/2$, threshold DSS scheme.

In section 5 we show how our techniques can be used to achieve adaptively-secure distributed protocols for other public-key systems. We show how to modify existing threshold RSA protocols (proven in the static model) to obtain adaptively-secure threshold RSA. Here also we achieve optimal-resiliency and constant-round protocols. We also show how our techniques allow for "upgrading" existent proactive discrete-log based threshold schemes from the static-adversary model to the adaptive one.

Related Work. Our work builds directly on previous protocols that were secure only in the static-adversary model, particularly on [Ped91b, GJKR99] for the distributed key generation and [GJKR96] for the threshold DSS signatures. We modify and strengthen these protocols to achieve adaptive security. Similarly, our solution to adaptive threshold RSA is based on the previous work of [Rab98].

As said before, if one is not concerned with the practicality of the solutions then general protocols for secure distributed computation of polynomial-time functions in the presence of adaptive adversaries are known. This was shown in [BGW88, CCD88] assuming (ideal) private channels. Later, [BH92] showed that with the help of standard encryption and careful erasure of keys one can translate these protocols into the model of public tappable channels. Recently, [CFGN96] showed how to do this translation without recurring to erasures but at the expense of a significant added complexity. Other recent work on the adaptive-adversary model includes [Can98, CDD$^+$99]. Also, independently from our work, adaptively-secure distributed cryptosystems have been recently studied in [FMY].

2 Technical Overview: Road-map to Adaptive Security

This section provides an overview of some basic technical elements in our work. It is intended as a high-level introduction to some of the issues that underly the protocol design and proofs presented in this paper (without getting into a detailed description of the protocols themselves, which are described in the later sections). We point out to some general aspects of our design and proof methodology, focusing on the elements that are essential to the treatment of the adaptive-adversary model in general, and to the understanding of our new techniques. For simplicity and concreteness, our presentation focuses mainly on threshold signature schemes which are threshold solutions to some existing centralized scheme like DSS or RSA. However, most of the issues we raise are applicable to other threshold functions and cryptosystems.

Threshold Signature Schemes. A threshold signature scheme consists of a distributed key generation protocol, a distributed signature generation protocol and a centralized verification algorithm (See the formal definitions in Appendix A). The signing servers first run the key generation, and obtain their private key shares and the global public key. Next, whenever a message needs to be signed, the servers invoke the signature generation protocol. The definition of secure threshold signature scheme makes two requirements. The first is *unforgeability*, which says, roughly, that, even after interacting with the signing parties in the initial generation of the distributed key and then in the signing protocol invoked on adaptively chosen messages, the adversary should be unable (except with negligible probability) to come up with a message that was not signed by the servers together with a valid signature. The second requirement is *robustness*, which says that whenever the servers wish to sign a message, a valid signature should be generated. Our model and definitions of Appendix A are a natural adaptation to the adaptive-adversary model of the definitions for threshold signatures found in [GJKR96], which in turn are based on the notion of security of a regular, centralized, signature scheme against chosen attacks as defined in [GMR88].

Proofs by Reduction and the Central Role of Simulations. We use the usual "reductionist" approach for the proofs of our protocols. Namely, given an adversary $\mathcal{A}$ that forges signatures in the distributed setting, we construct a forger $\mathcal{F}$ that forges signatures of the underlying centralized signature scheme. Thus, under the assumption that the centralized scheme is secure, the threshold signature scheme must also be secure. A key ingredient in the reduction is a "simulation" of the view of the adversary $\mathcal{A}$ in its run against the distributed protocol.[1] That is, the forger $\mathcal{F}$ builds a virtual distributed environment where the instructions of $\mathcal{A}$ are carried out. Typically, first $\mathcal{F}$ has to simulate to $\mathcal{A}$ an execution of the distributed key generation that results in the same public key against which $\mathcal{F}$ stages a forgery. Then $\mathcal{A}$ will successively invoke the distributed signature protocol on messages of his choice, and $\mathcal{F}$, having access to a signing oracle of the underlying centralized signature scheme, has to simulate to $\mathcal{A}$ its view of an execution of the distributed signature protocol on these messages. Eventually, if $\mathcal{A}$ outputs a forgery in the imitated environment, $\mathcal{F}$ will output a forgery against the centralized signature scheme. Two crucial steps in the analysis of this forger are: (1) Demonstrate that the adversary's view of the simulated interaction is indistinguishable[2] from its view of a real interaction with parties running the threshold scheme. (2) Demonstrate that the forger can translate a successful forgery by the adversary (in the simulated run) to a successful forgery of the centralized signature scheme.

The first step is the technical core of our proofs (see Lemmas 1 and 2). Furthermore, to carry out the second step we need simulators that are able to generate views that are indistinguishable from the view of the adversary *under a given conditioning.* More specifically, $\mathcal{F}$ has to simulate a run of the distributed key generation that arrives at a *given* public key of the underlying centralized scheme, and it has to simulate a run of the distributed signature scheme that arrives at a *given* signature output by the signing oracle of the centralized scheme. We refer to this as the problem of *hitting* a particular value in the simulation.

Problems with Simulating an Adaptive Adversary. The above proof technique is also used in the case of static adversaries. However, an adaptive adversary can corrupt any player at any time (as long as not too many parties are corrupted) and at that point the simulator needs to be able to provide the attacker with the current internal state of the broken party. In particular, this information must be consistent with the information previously seen by the attacker. Providing this information is typically the main difficulty in proving adaptive security of protocols. In paragraph "adaptive vs. non-adaptive" of Section 3, we show that a simulation of previous threshold

[1] The technique of simulation of adversary's view in a protocol was introduced by [GMR89].

[2] The notion of computational indistinguishability was introduced in [Y82] and [GM84].

protocols, provably static-resistant, fails in this way in the adaptive setting. This does not necessarily mean that previous solutions are insecure in the presence of an adaptive attacker, but it shows, at least, that current analytical techniques seem insufficient to prove their security in this stronger model.[3]

Here we demonstrate the difficulty of simulating a view of the adaptive adversary with a simplified example taken from the protocol discussed in the aforementioned paragraph: Let p, q be large primes s.t. $q|(p-1)$ and g an element of order q in Z_p^*. Assume that each server P_i holds a random secret value $x_i \in Z_q$ such that values $x_1, ..., x_n$ encode, via Shamir polynomial secret sharing of degree t, a secret quantity $x \in Z_q$, such as a signing key.[4] We want the players to compute the public key $y = g^x \bmod p$ by having each player P_i make public (broadcast) a value $g^{x_i} \bmod p$, from which y can be computed via interpolation in the exponent (namely, for any group G of $t+1$ players, $y = g^x = \prod_{i \in G}(g^{x_i})^{\lambda_i} \bmod p$, where λ_i's are the known Lagrange coefficients such that $x = \sum_{i \in G} \lambda_i x_i \bmod q$). To prove that the adversary learns no other information about x throughout the protocol except g^x, we have to show a simulator which generates the adversary's view of this protocol with the knowledge of only the public key $y = g^x$. Specifically, the simulator must produce values $g^{x_1}...g^{x_n}$ which agree with $y = g^x$, and values x_i for the players corrupted by the adversary. However, an adaptive adversary might not corrupt any players initially, and decide which ones to corrupt only after seeing the broadcasted g^{x_i} values. He will then expect to see the secret exponents x_i of the corrupted servers being consistent with the g^{x_i} values that he has just seen. Since the x_i's encode x (which is unknown to the simulator) via Shamir secret sharing with a polynomial of degree t, the simulator cannot know more than t of these values. Indeed, the only way to produce the g^{x_i}'s that agree with $y = g^x$ while maximizing the number of players for which the simulator can reveal a correct value x_i is the following: The simulator picks a t element subset $T \subset \{1, ..., n\}$ of players, picks values x_i, $i \in T$ at random, computes the corresponding g^{x_i}'s, and uses these values and y to compute the remaining $n-t$ values g^{x_i} via interpolation in the exponent. Since the simulator cannot predict which servers will be corrupted, the chances that the subset T he picks is going to include all the servers which the adversary decides to corrupt are negligible. In contrast, the simulation of such a protocol *is* possible in the case of a static attacker where the simulator knows in advance the set of corrupted players, because then the simulator simply picks the subset T that includes all these players.

Erasures. One way to get around the above problem in the adaptive-adversary model is to specify in the protocol that the servers *erase* the private values x_i before the values g^{x_i} are broadcasted. Now, when corrupting P_i, this information is not available to the adversary in the real run of the protocol and therefore there is no need to provide it in the simulation. However, this technique can only be applied when the protocol no longer needs x_i. A careful use of erasures is at the core of the design of our protocols. In some cases, this requires that information that could have been useful for the continuation of the protocol be erased. Two examples of crucial erasures in our protocols are the erasure of some of the verification information kept by Pedersen's VSS protocol [Ped91a] (which we compensate for with the use zero-knowledge proofs – see below), and the erasure of all temporary information generated during each execution of the signature protocol. Furthermore, erasures simplify the task of implementing private channels with conventional encryption in the adaptive model (see below).[5]

Rewinding the Adversary. Another useful technique for getting around the problem of having to present the attacker with information not available to the simulator is rewinding. This is a well-known technique for proving zero-knowledge (and other) protocols. In its essence it allows a simulator that is "in trouble" with its simulation to rewind the adversary's state to a previous computation state, and restart the computation from there. At this point the simulation will try some other random choices in the computation hoping that it will not end in another bad position as before. Thus, in the case where the chances to get "stuck" again are not too large then rewinding is a very useful technique. In a second try, after the simulator makes some different choices, the adversary will hopefully not hit an "inconsistent" situation again.

[3] As mentioned in the introduction, resistance to static adversaries does not imply, in general, resistance to adaptive ones. For example, [CDD+99] have recently shown that a known protocol is secure against static attackers but is insecure against an adaptive attacker. (Interestingly, this protocol resists even computationally-unlimited static attackers but succumbs to computationally-bounded adaptive attackers.)

[4] In other words, there exists a t-degree polynomial $f(z)$ s.t. $x_i = f(i) \bmod q$ for all $i \in \{1, ..., n\}$ and $x = f(0) \bmod q$. Therefore, the random values $x_1, ..., x_n$ are t-wise independent.

[5] Successful erasure of data is not a trivial task. Extra precaution is needed to insure that the data, and all its back-ups, are carefully overwritten (one needs to watch out for long-term 'residual memory' of magnetic storage media, for untrustworthy operating systems, etc). In our setting, we trust *uncorrupted* parties to properly erase data whenever required by the protocol. This is somewhat similar to the requirement that parties erase Diffie-Hellman exponents to achieve perfect forward secrecy [DOW92]. (See [CFGN96, Can98] for further discussion on the issue of data erasures.)

Note that rewinding is a proof technique, not a protocol action. Yet, one has to design the protocol in ways that make rewinding a useful tool. As in the case of erasures, correct use of rewinding requires care. Following is an important instance where an improper use of rewinding can render the whole simulation useless.[6] Assume that the distributed adversary $\mathcal{A}$ asks for signatures on a sequence of messages $m_1, m_2, \ldots$. For each message m_i, in order to simulate the signature protocol, the forger $\mathcal{F}$ must ask the signing oracle of the underlying centralized signature scheme for a signature on m_i. Assume now that during the k^{th} signature protocol (while simulating the distributed signing of message m_k) the simulator gets "stuck" and needs to rewind the adversary back to a *previous* signature protocol, say the j^{th} one for $j < k$. Now the adversary is rewinded back to m_j, the subsequent view is different, so with all likelihood the adversary will start asking for signatures on *different* messages. In other words the sequence of messages is now $m_1, \ldots, m_j, m'_{j+1}, \ldots, m'_k, \ldots$. However the sequence of messages asked by the forger $\mathcal{F}$ to the signature oracle is $m_1, \ldots, m_j, m_{j+1}, \ldots, m_k, m'_{j+1}, \ldots, m'_k$, i.e. the forger has now asked *more* messages than the adversary (indeed since the adversary was rewinded he has no recollection of asking the messages $m_{j+1}, \ldots, m_k$). This means that the adversary $\mathcal{A}$ may output one of those messages as a successful forgery, but such event will not count as a success for the forger $\mathcal{F}$.

It is important then to confine rewinding of the adversary *inside* a simulation of a single run of the signature protocol. That is, the simulator *must not* be allowed to rewind an adversary to a previous signature protocol. One of the tools that we use to ensure that our protocols are simulatable in such a way is the erasure of local temporary information generated by our protocols.

The Single-Inconsistent-Player Technique. Another concern that needs to be addressed is making sure that rewindings do not take place too often (otherwise the simulator may not run in polynomial time). Very roughly, we guarantee this property as follows. We make sure that the simulator can, at any point of the simulated run of the protocol, present the attacker with a correct internal state (i.e. state that is consistent with the attacker's view) for all honest players except, maybe, for *one server.* This server is chosen by the simulator at random among the currently non-corrupted servers. Moreover, the view of the attacker is independent from this choice of the simulator. This guarantees that the inconsistent server is corrupted (which forces the simulator to stop and rewind the simulation) with probability at most one half (this probability is given by the ratio t/n of corrupted players). Thus, the expected number of rewindings is at most one. In the "Adaptive vs. Non-Adaptive Solutions" paragraph of Section 3, we show how we used this technique in our protocols. Namely, we show that the crucial modification which created an adaptively-secure key generation protocol from a previous protocol of [GJKR99] which was secure in only the static-adversary model, was the reduction of the number of "inconsistent" players to only one.

Zero-Knowledge to the Rescue, and Efficiently. We have explained the importance of having during simulation at most a single player with possible inconsistent internal state. In our protocols, this property is achieved, to a large extent, using erasure of internal information by the players that otherwise could create inconsistencies with the attacker's view. Usually, the erased information was useful for verifying future actions by other players, and thus its erasure forces us to find other ways for verification of these actions in the protocol. We turn for this task to zero-knowledge proofs [GMR89], and more specifically to zero-knowledge proofs of knowledge.

Useful as they are, zero-knowledge proofs may add significant complexity to the protocols, degrade performance, and increase communication. We show how to make intensive use of zero-knowledge proofs with significant savings in complexity. Specifically, we show how to achieve the effect of $O(n^2)$ zero-knowledge proofs of knowledge (where each of the n players proves something to each of the other players) in a *single* 3-move *honest verifier* zero-knowledge proof. This is done by implementing the honest verifier using a distributed generation of a challenge by all players.[7] We implement this technique for Schnorr's proof of possession of discrete-log [Sch91]; the same technique can be used for other zero-knowledge protocols as well [CD98].

Maintaining Private Channels in the Adaptive Model. An important observation about the design of our protocols is that we specify them using the abstraction of "private channels" between each pair of parties. This

[6] We remark that the rewinding technique may also cause difficulties when the signature protocol is composed with other protocols, and in particular when several copies of the signature protocol are allowed to run *concurrently.* We leave these issues out of the scope of this work. See more details in [DNS98, KPR98].

[7] Recall that 3-move zero-knowledge proofs cannot exist for *cheating* verifiers if the underlying problem is not in BPP [GK96, IS93]. Thus, the distributed nature of the verifier in our implementation is essential for "forcing honesty". In particular, our simulation of these proofs does not require rewinding at all.

is a usual simplifying approach in the design of cryptographic protocols: The underlying assumption is that these private channels can be implemented via encryption. In the case of adaptive security, however, this simple paradigm needs to be re-examined. Indeed, a straightforward replacement of private channels with encryption could ruin the adaptive security of our protocols. Fortunately, in settings where data erasures are acceptable (and in particular in our setting) a very simple technique exists [BH92] for solving this problem. It involves local refreshment of (symmetric) encryption keys by each party, using a simple pseudorandom generator and without need for interaction between the parties. This adds virtually no overhead to the protocols beyond the cost of symmetric encryption itself.

3 Adaptively-secure Distributed Key Generation

A basic component of threshold cryptosystems based on the difficulty of computing discrete logarithms is the shared generation of a secret x for which the value g^x is made public. Not only is this needed to generate a shared key without a trusted dealer but it is also a sub-module of other protocols, e.g. as used in our own threshold DSS scheme for generating $r = g^{k^{-1}}$ (see Sec. 4). We call this module a "Distributed Key Generation for Discrete–Log Based Cryptosystems" (DL-Key-Gen) protocol. This primitive is defined in [GJKR99], yet the solution provided in that paper is proven secure only against a non-adaptive adversary. A DL-Key-Gen protocol is performed by n players $P_1, ..., P_n$ on public input (p, q, g) where p and q are large primes, q divides p–1, and g is an element of order q in Z_p^*.[8] DL-Key-Gen generates a Shamir secret sharing of a uniformly distributed random secret key $x \in Z_q$, and makes public the value $y = g^x \bmod p$. At the end of the protocol each player has a private output x_i, called a *share* of x (each group of $t + 1$ shares of the uncorrupted players defines the same value x). The protocol is secure if in the presence of an adversary who corrupts at most t players, the protocol generates the desired outputs and does not reveal any information about x except for what is implied by the public value $g^x \bmod p$.[9]

To ensure the ability to use the DL-Key-Gen protocol as a module in a larger, adaptively secure, threshold scheme (like the DSS-ts signature scheme of Section 4) we add to the requirements of [GJKR99] that at the end of the protocol each party must *erase* all the generated internal data pertaining to this execution of the protocol except of course for its private output x_i. This requirement ensures that the simulator of the threshold scheme within which DL-Key-Gen is used as a module (e.g. DSS-Sig-Gen and the DSS-ts scheme as a whole, see section 4) is able to produce the internal state of a player whom the adversary corrupts *after* the execution of the DL-Key-Gen module is completed.

Distributed Key Generation Protocol. We present a distributed key generation protocol DL-Key-Gen with resilience $t < n/3$, which is simple to explain and already contains the design and analysis ideas in our work. (See Figure 3 below for the modifications required to achieve optimal resilience $t < n/2$.) DL-Key-Gen is based on the Distributed Key Generation protocol of [GJKR99] (called DKG there) which is proven secure only against a non-adaptive adversary. (Some of the changes made to this protocol in order to achieve adaptive security are discussed below.)

Protocol DL-Key-Gen presented in Fig.1, starts with inputs (p, q, g, h) where (p, q, g) is a discrete–log instance and h is a random element in the subgroup of Z_p^* generated by g. When DL-Key-Gen is executed on inputs (p, q, g) only (which happens when the DL-Key-Gen is invoked for the first time), a random element h in the subgroup of Z_p^* generated by g, must first be publicly generated as follows: If q^2 does not divide $p - 1$, the players

[8] For technical reasons, we additionally require that p–1 is *not* divisible by q^2 (however, p–1 *is* divisible by q).

[9] Compare Appendix A, where we provide a syntactic definition of distributed key generation Key-Gen. DL-Key-Gen is an example of Key-Gen for specific discrete–log based signature schemes like DSS (x_i's here are the private outputs s_i of Key-Gen, and y, together with (p, q, g), is the public output v). However, in Appendix A we assume that the only input to Key-Gen is a security parameter k, while DL-Key-Gen protocol we describe here starts with public values (p, q, g) which satisfy specific properties.

However, the definition of security of a threshold signature scheme in Appendix A can be modified so that the distributed key generation protocol Key-Gen starts with more public input than just the security parameter k. For example, the security of a DSS scheme (see Appendix C) is claimed for *every* prime p, prime q, and generator g of specified length. Therefore, the (p, q, g) triple can be a public input to DL-Key-Gen, because it can be centrally generated by any party, and its required properties can be verified by all participating players.

However, in general, the (p, q, g) values can also be easily publicly generated with randomized procedures that find primes and elements of a given prime order, using randomness jointly generated by the players via a generic coin-flipping protocol ([BGW88] for $t < n/3$ or [CDD+99] for $t < n/2$).

Input: Parameters (p, q, g), and h an element in the subgroup generated by g
Public Output: y the public key
Secret Output of P_i: x_i the share of the random secret x
(all other secret outputs are erased)
Other Public Output: Public commitments

[If h is not passed as an input (first execution of DL-Key-Gen):
Generating h: Players generate a random $r \in Z_p^*$ via a collective coin-flipping protocol [BGW88]. Set h to $r^{(p-1)/q} \bmod p$.]

Generating x:
Players execute Joint-RVSS(t)

1. Player P_i gets the following secret outputs of Joint-RVSS:
   – x_i, x'_i his share of the secret and the associated random value (resp.)
   – $f_i(z), f'_i(z)$ polynomials he used to share his contribution $z_i = f_i(0)$ to x
   – s_{ji}, s'_{ji} for $j = 1..n$, the shares and the associated randomness he received from others
   Players also get public outputs C_{ik} for $i = 1..n,\ k = 0..t$, and the set $QUAL$.

Extracting $y = g^x \bmod p$:
Each player exposes $y_i = g^{z_i} \bmod p$ to enable the computation of $y = g^x \bmod p$.

2. Each player P_i, $i \in QUAL$, broadcasts $A_i = g^{f_i(0)} = g^{z_i} \bmod p$ and $B_i = h^{f'_i(0)} \bmod p$, s.t. $C_{i0} = A_i B_i$. P_i also chooses random values r_i and r'_i and broadcasts $T_i = g^{r_i}, T'_i = h^{r'_i} \bmod p$.
3. Players execute Joint-RVSS(t) for a joint random challenge d. Player P_i sets his local share of the secret challenge to d_i. All other secret output generated by this Joint-RVSS and held by P_i is erased.
4. Each player broadcast d_i. Set $d =$ EC-Interpolate$(d_1, \ldots, d_n)$.
5. P_i broadcasts $R_i = r_i + d \cdot f_i(0)$ and $R'_i = r'_i + d \cdot f'_i(0)$
6. Player P_j checks for each P_i that $g^{R_i} = T_i \cdot A_i^d$ and $h^{R'_i} = T'_i \cdot B_i^d$. If the equation is not satisfied then P_j complains against P_i.
7. If player P_i receives more than t complaints, then P_j broadcasts s_{ij}. Set $z_i =$ EC-Interpolate$(s_{i1}, \ldots, s_{in})$ and $A_i = g^{z_i}$.
8. The public value y is set to $y = \prod_{i \in QUAL} A_i \bmod p$.
9. Player P_i erases all secret information generated in this protocol aside from his share x_i.

Figure 1: DL-Key-Gen - Distributed Key Generation, $n \geq 3t + 1$

generate a random $r \in Z_p^*$ via a collective coin-flipping protocol [BGW88] and take $h = r^{(p-1)/q} \bmod p$.[10] The protocol proceeds as follows: The first part of generating x is achieved by having each player commit to a random value z_i via a Joint Pedersen's VSS [Ped91a, Ped91b, GJKR99] (Figure 7, Appendix B). These commitments are verified by the other players and the set of parties passing verification is denoted by $QUAL$. Then the shared secret x is set (implicitly) to $x = \sum_{i \in QUAL} z_i \bmod q$. In addition to enabling the generation of a random, uniformly distributed value x, Joint-RVSS has the side effect of having each player P_i broadcast an information-theoretically private commitment to z_i of the form $C_{i0} = g^{z_i} h^{f'_i(0)} \bmod p$, where f'_i is a randomizing polynomial chosen by P_i in the run of Joint-RVSS. We utilize these commitments to "extract" and publish the public key $y = g^x \bmod p$ in the following way:

We have that $g^x = g^{\sum_{i \in QUAL} z_i} = \prod_{i \in QUAL} g^{z_i} \bmod p$. Thus, if we could have each player "deliver" g^{z_i} in a verifiable way then we could compute y. To that end, we require P_i to "split" his commitment C_{i0} into the two components $A_i = g^{z_i}$ and $B_i = h^{f'_i(0)}$ (Step 2). To ensure that he gives the correct split, P_i proves that he knows both $Dlog_g A_i$ and $Dlog_h B_i$ with Schnorr's 3-round zero-knowledge proof of knowledge of discrete-log [Sch91]. We exploit the fact that each player is proving his statement to many verifiers by generating a single challenge

[10] We chose to write DL-Key-Gen with h as an additional input, so that it could be invoked as a module by the DSS-ts scheme of Section 4 without generating h each time.

for all these proofs. Joint challenge d is generated with Joint-RVSS with public reconstruction (Steps 3-4). The proof completes in Steps 5-6. For the interesting properties of this form of zero-knowledge proof see Section 2. If a player fails to prove a correct split, then his value z_i is reconstructed using polynomial interpolation with error-correcting codes.[11] The players then *erase* all the information generated during the protocol except of their share of the secret key.

Proving Adaptive Security of Key Generation. In Figure 2 we present a simulator SIM-DL-Key-Gen for the DL-Key-Gen protocol.[12] This simulation is the crux of the proof of secrecy in the protocol, namely, that nothing is revealed by the protocol beyond the value $y = g^x \bmod p$ (to show this we provide the value of y as input to the simulator and require it to simulate a run of the DL-Key-Gen protocol that ends with y as its public output). We denote by $\mathcal{G}$ (resp. $\mathcal{B}$) the set of *currently* good (resp. bad) players. The simulator executes the protocol for all the players in $\mathcal{G}$ except one. The state of the special player P (selected at random) is used by the simulator to "fix" the output of the simulation to y, the required public key. Since the simulator does not know $Dlog_g y$ it does not know some secret information relative to this special player (in particular the component z_P that this player contributes to the secret key). This lack of knowledge does not disable the simulator from proving that it knows P's contribution (Steps 2-6 in DL-Key-Gen) since the simulator can utilize a simulation of this ZK proof. However, if the adversary corrupts P during the simulation (which happens with probability $< 1/2$) the simulator will not be able to provide the internal state of this player. Thus, the simulator will need to rewind the adversary and select another special player P'. The simulation will conclude in expected polynomial time.

Lemma 1 *Simulator* SIM-DL-Key-Gen *on input* (p, q, g, h, y) *ends in expected polynomial time and computes a view for the adversary that is indistinguishable from a view of the protocol* DL-Key-Gen *on input* (p, q, g, h) *and output* y.

Proof: First we show that SIM-DL-Key-Gen outputs a probability distribution which is *identical* to the distribution the adversary sees in an execution of DL-Key-Gen that produces y as an output. In the following denote by G_g the subgroup of Z_p^* generated by g.

1. The first step is carried out according to the protocol, thus values $f_i(j), f'_i(j)$, $i \in \mathcal{G}$, $j \in \mathcal{B}$, and C_{ik}, $i \in \mathcal{G}$, $k = 0 \dots t$ have the required distribution.
2. The values A_i for $i \in \mathcal{G} \setminus \{P\}$ are distributed exactly as in the real protocol. The value $A_P^* = y \cdot \prod_{i \in (QUAL \setminus \{P\})} (A_i)^{-1} = y \cdot \prod_{i \in (\mathcal{G} \setminus \{P\})} (A_i)^{-1} \cdot \prod_{i \in (\mathcal{B} \cap QUAL)} (A_i)^{-1}$ is distributed uniformly in G_g and independently from the values A_i for all i. This is because y is random and uniformly distributed and the A_i for $i \in \mathcal{B}$ are generated independently from the other ones (because Joint-RVSS is information theoretically private).

   A similar argument holds for values B_i, $i \in \mathcal{G} \setminus \{P\}$ and B_P^*.

   Finally, values T_i, T'_i, $i \in \mathcal{G} \setminus \{P\}$ are picked at random in G_g as in the protocol. Values $T_P^*, T_P^{'*}$ are also uniformly distributed in G_g. Thus the view in this step is identical.
3. Here SIM-DL-Key-Gen performs a Joint-RVSS(t) protocol on behalf of the players in $\mathcal{G}$ exactly as in the protocol. Thus the view in this step is identical.
4. In this step SIM-DL-Key-Gen broadcasts shares d_i^* of a new polynomial $f_d^*(z)$ which is random subject to the constraint that $f_d^*(j) = f_d(j)$ for $j \in \mathcal{B}$ and $f_d^*(0) = d$. Although these d_i^*'s are not the same values held as shares by the players in the previous step, the view for the adversary is still the same as in the real protocol. This is because the adversary has seen a Joint-RVSS of value d' and at most t points of the sharing polynomial f. It is easily seen that for any other value d there is another polynomial that

[11] We use notation $v = \text{EC-Interpolate}(v_1, \dots, v_n)$ for public reconstruction of a value through polynomial interpolation with the use of error-correcting codes. If $\{v_1, \dots, v_n\}$ is a set of values, such that at least $n - t$ of them lie on some d-degree polynomial $F(\cdot)$, and $n > d + 2t$, then $v = \text{EC-Interpolate}(v_1, \dots, v_n) = F(0)$. The polynomial can be computed using any standard error-correction mechanism, e.g. the Berlekamp-Welch decoder [BW]. In our applications it will be either $d = t$ (Steps 4 and 7, Figure 1) or $d = 2t$ (Steps 1e and 2, Figure 4).

[12] If the simulator simulates the first execution of DL-Key-Gen when it is invoked on p, q, g only, and no h, the simulator needs to simulate the view of the h-generation. It does it by simply playing the part of the honest players in that protocol.

Input: public key y and parameters (p, q, g) and h an element generated by g

[If h is not passed as an input (simulation of first execution of DL-Key-Gen):
Perform the h-generation protocol on behalf of the players in $\mathcal{G}$.]

1. Perform Step 1 of DL-Key-Gen on behalf of the players in $\mathcal{G}$. At the end of this step the set $QUAL$ is defined. SIM-DL-Key-Gen knows all polynomials $f_i(z), f'_i(z)$ for $i \in QUAL$ (as it controls a majority of the players). In particular, SIM-DL-Key-Gen knows the values $f_i(0), f'_i(0)$.

Perform the following pre-computations:
– Choose at random one uncorrupted player $P \in \mathcal{G}$
– Compute $A_i = g^{f_i(0)}, B_i = h^{f'_i(0)}$ for $i \in QUAL \setminus \{P\}$
– Set $A^*_P = y \cdot \prod_{i \in (QUAL \setminus \{P\})} (A_i)^{-1} \bmod p$, and $B^*_P = C_{P0}/A^*_P \bmod p$
– Pick values $d, R_P, R'_P \in_R Z_q$, set $T^*_P = g^{R_P} \cdot (A^*_P)^{-d}$ and $T'^*_P = h^{R'_P} \cdot (B^*_P)^{-d}$

2. For each player $i \in \mathcal{G} \setminus \{\mathcal{P}\}$ execute Step 2 according to the protocol. For player P broadcast $A^*_P, B^*_P, T^*_P, T'^*_P$ which were computed previously.
3. Perform the Joint-RVSS(t) protocol on behalf of the uncorrupted players. Note that SIM-DL-Key-Gen knows the shares d_i, $i \in \mathcal{B}$. *Erase* all the secret output of the uncorrupted players in this protocol. Pick a t-degree polynomial $f^*_d(z)$ s.t. $f^*_d(0) = d$ and $f^*_d(i) = d_i$ for $i \in \mathcal{B}$. Set $d^*_i = f^*_d(i)$ for $i \in \mathcal{G}$.
4. Broadcast d^*_i for each $i \in \mathcal{G}$.
5. Broadcast $R_i = r_i + d \cdot f_i(0) \bmod q$ and $R'_i = r'_i + d \cdot f'_i(0) \bmod q$ for $i \in \mathcal{G} \setminus \{P\}$ and R^*_P, R'^*_P for player P.
6. Execute Step 6 of DL-Key-Gen for all players. (Notice that the corrupted players can publish a valid complaint only against one another.)
7. For each player with more than t complaints participate in the reconstruction of their value. Note that only players in $\mathcal{B}$ can have more than t complaints.
8. Erase all information aside from the value x_i.

Figure 2: SIM-DL-Key-Gen - Simulator for the DL-Key-Gen Protocol

passes through the t points held by the adversary and the free term d. Notice that since only the d^*_i's are broadcasted the simulator does not have to "match" the public commitments generated by the Joint-RVSS.

5. Values $R_i, R'_i, i \in \mathcal{G}$ satisfy the required constraints (i.e. the verification equation) as in the protocol.

We have shown that the *public* view of the adversary during the simulation is identical to the one he would see during a real execution. Now we must proceed to show that the simulator can produce a consistent view of the internal states for the players corrupted by the adversary $\mathcal{A}$. Clearly, if a player is corrupted before Step 2, the simulator can produce a consistent view because it is following the protocol. After Step 2 the simulator can show correct internal states for all the players in $\mathcal{G}$ except for the special player P. Thus, if P is corrupted the simulator rewinds the adversary to the beginning of Step 2 and selects at random a different special player. Notice that if a player P_i in $\mathcal{G} \setminus \{P\}$ is corrupted after Step 4, the simulator has broadcasted for P_i a "fake" value d^*_i. But since we erased all the private information (except the shares) generated in Joint-RVSS in Step 3, the simulator can simply claim that d^*_i was really the share held by P_i. This will not contradict any of the generated public information. □

Notice that at the end of the protocol the simulator erases *all* the secret information generated during this protocol aside from x_i. Thus if a player in $\mathcal{G}$ (*including* the special player P) is corrupted after Step 9 the simulator simply provides the adversary with x_i and claims that all the rest of the information has been erased.

Key Generation with Optimal Resilience. We describe the changes required to achieve an optimally-resilient (i.e. $n = 2t + 1$) DL-Key-Gen protocol in Figure 3. The modification described in item 2 of that figure must be reflected in the simulator, because the current SIM-DL-Key-Gen is unable to produce in Step 4 the random

1. The generation of h which occurs when DL-Key-Gen is executed for the first time is performed with an adaptively-secure VSS with an optimal $t < n/2$ resilience (e.g. [CDD$^+$99]), instead of with a VSS protocol which has a $t < n/3$ resilience ([BGW88]).
2. In DL-Key-Gen of Figure 1, we publicly reconstruct a value created with Joint-RVSS protocol (see Steps 3-4). This reconstruction is done using error-correcting codes, which makes the protocol easy to simulate, but which tolerates only $t < n/3$ faults. However, we can achieve optimal resilience by sieving out bad shares with Pedersen verification equation (Eq.(1)) if the players broadcast in Step 4 the associated random values, together with their shares, generated by the Joint-RVSS of the previous step. Therefore the players must no longer erase these values as in the current Step 3.
3. In Step 9 the players erase all their secret information but their shares x_i as before, except that they hold on also to the associated random values x'_i.

Figure 3: Optimally-resilient, $n \geq 2t + 1$, DL-Key-Gen Protocol (changes from Figure 1)

associated values produced in Joint-RVSS of Step 3 (Figure 1). However, the simulator could produce them if he knew the discrete logarithm $\lambda = Dlog_g(h)$. (Note that SIM-DL-Key-Gen in Step 3 knows the shares d_i and the random associated values d'_i for all i. Knowing λ, SIM-DL-Key-Gen can compute the random associated values d_i^*' for any d_i^* of his choice by solving $d_i + \lambda d'_i = d_i^* + \lambda d_i^*$'.) Therefore, in the simulation of the h-generation protocol, instead of playing the part of the honest parties, the simulator must pick $\lambda \in Z_q$ at random, compute $h = g^\lambda$, simulate the VSS protocol of [CDD$^+$99] to arrive at $r = h^\beta$ where $\beta = \lambda((p-1)/q)^{-1} \bmod q$, and use λ in the modified simulation of Steps 3-4 as described above.

Adaptive vs. Non-Adaptive Solutions. As noted before, DL-Key-Gen is based on the recent distributed key generation protocol of [GJKR99], which is secure only against a static adversary. The generation of x is the same in both protocols but they differ in the method for extracting the public key. In the current protocol each player reveals only values $(A_i, B_i) = (g^{f_i(0)}, h^{f'_i(0)})$ from the execution of Joint-RVSS, and uses zero-knowledge proofs to guarantee that these values are properly formed. Due to this limited revealing of information it is sufficient for the simulator to "cheat" with respect to the internal state of only a single player, and yet to "hit" the desired value. However, in the protocol of [GJKR99], each player reveals A_i by publishing all values $g^{a_{ik}}$, $k = 0, ..., t$. For one of the players the simulator has to commit in this way to a polynomial without knowing $a_{i0} = f_i(0)$. Therefore he can do it in a way that is consistent with only t points on this polynomial. Thus, the simulator has an inconsistent internal state for $n - t$ players, and hence has to stop and rewind every time one of them is corrupted, which happens with overwhelming probability if the adversary is adaptive.

4 Adaptively-secure Threshold DSS Signature Scheme

As described in Section 2, a Threshold Signature Scheme consists of a distributed key generation, distributed signature protocol, and a signature verification procedure (see full definitions in Appendix A). Here we present an adaptively-secure Threshold DSS Signature Scheme (see Appendix C for the description of a regular, centralized DSS signature scheme). We first present a distributed DSS signature protocol DSS-Sig-Gen with $t < n/4$ resilience. Then we prove the unforgeability and robustness of the resulting Threshold DSS Signature Scheme DSS-ts which is combined from DL-Key-Gen of Section 3, DSS-Sig-Gen, and the regular DSS verification procedure which we will here denote DSS-Ver. Finally we describe how the optimal-resilient ($t < n/2$) version of this signature protocol and the resulting DSS signature scheme. We refer the reader to Section 2 for a higher-level description of the basic elements in our approach, solutions, and proofs.

Distributed Signature Protocol. The basis for the signature protocol DSS-Sig-Gen (Fig.4) is the protocol of [GJKR96] (see that paper for the full rationale of this protocol), with modifications to allow for the adaptive adversary. Protocol DSS-Sig-Gen assumes that the signing parties have previously executed the DL-Key-Gen protocol and hold the shares x_i of a secret key that corresponds to the generated public key y. It also assumes that the public input includes p, q, g, h, all generated before DL-Key-Gen started.

The protocol DSS-Sig-Gen is invoked every time some message m needs to be signed.

Input: message m to be signed (plus the outputs of DL-Key-Gen)
Public Output: (r, s) the signature of the message m

1. **Generate** $r = g^{k^{-1}} \bmod p \bmod q$
   (a) Generate k. Players execute Joint-RVSS(t). Player P_i sets k_i to his share of the secret. All other secret information generated by the above execution is erased.
   (b) Generate random sharings of 0 on polynomials of degree $2t$. Players execute two instances of Joint-ZVSS$(2t)$. Player P_i sets b_i and c_i to his shares of the two sharings. All other secret information generated by the above executions is erased.
   (c) Generate a random value a and $g^a \bmod p$ using DL-Key-Gen. Player P_i sets a_i to his share of the secret a.
   (d) Player P_i broadcasts $v_i = k_i a_i + b_i \bmod q$.
   (e) Each player computes: $\mu \stackrel{\triangle}{=}$ EC-Interpolate$(v_1, \ldots, v_n) \bmod q$ $[= ka \bmod q]$, then $\mu^{-1} \bmod q$, and $r \stackrel{\triangle}{=} (g^a)^{\mu^{-1}}$ $[= g^{k^{-1}}] \bmod p \bmod q$.
2. **Generate** $s = k(m + xr) \bmod q$
   P_i broadcasts $s_i = k_i(m + x_i r) + c_i \bmod q$. Set $s \stackrel{\triangle}{=}$ EC-Interpolate$(s_1, \ldots, s_n)$.
3. Player P_i erases all secret information generated in this protocol.

Figure 4: DSS-Sig-Gen - Distributed Signature Generation, $n \geq 4t + 1$

The first part of the signature computation is the generation of the random value $r = g^{k^{-1}} \bmod p \bmod q$. This computation is very similar to the distributed key generation protocol, aside from the complication that it requires g to be raised to the inverse of the shared secret value k. We achieve this with a variation of the distributed inversion protocol from [BB89]: The players select a random uniformly distributed $k \in Z_q$ in shared form via the Joint-RVSS protocol (Step 1a). Then they perform a DL-Key-Gen protocol to select a random $a \in Z_q$ in shared form and publish $g^a \bmod p$ (Step 1c). The inversion of k in the exponent occurs when the players reconstruct in the clear the product $\mu = ka \bmod q$ which is a random number, invert it, and then compute $g^{k^{-1}} = (g^a)^{\mu^{-1}}$ (Steps 1d-1e). The value s is publicly reconstructed when each player reveals the product $k_i(m + x_i r)$ which lies on a $2t$-degree polynomial whose free term is $s = k(m + xr)$. In order to hide all partial information during the public reconstruction of values μ and s (Steps 1e and 2), we use the Joint-ZVSS subprotocol (see Section B, also Figure 7) twice to generate two randomizing polynomials (Step 1b) that are added to the polynomials that secret-share μ and s respectively. As in DL-Key-Gen, it is crucial for the proof of adaptive security that the players erase at the end all secret information generated by this protocol.

Proving Adaptive Security of Threshold DSS. We state the claim about the security (see definitions of security in Appendix A) of the threshold signature (ts) scheme DSS-ts= (DL-Key-Gen, DSS-Sig-Gen, DSS-Ver):

Theorem 1 *If DSS is unforgeable under adaptive chosen message attack then* DSS-ts *is a secure (unforgeable and robust) t-threshold* ts *scheme for $t < n/4$.*

Proof: This follows from the unforgeability and robustness lemmas (Lemma 3 and 4) which we prove below. Note that if DSS is unforgeable under adaptive chosen message attack then in particular computer discrete logarithms is hard, and Lemma 4 follows. □

To prove unforgeability, we first need a lemma about the simulator SIM-DSS-Sig-Gen presented in Fig.5:

Lemma 2 *Simulator* SIM-DSS-Sig-Gen *on input $(p, q, g, h, y, m, (r, s))$ ends in expected polynomial time and computes a view for the adversary that is indistinguishable from a view of the protocol* DSS-Sig-Gen *on input public key (p, q, g, h, y), message m, and output signature (r, s).*

Input: message m, its signature (r, s) of a DSS system (y, p, q, g),
an element h generated by g

Compute $r^* = g^{ms^{-1}} y^{rs^{-1}} \bmod p$. Pick $\mu \in_R Z_q$, and set $\beta = (r^*)^\mu \bmod p$.

1. (a) Execute Joint-RVSS(t) on behalf of the uncorrupted players. This results in shares k_i for each of the uncorrupted players. Erase all secret values aside from k_i.
   (b) Execute two sharings of a 0 value on polynomials of degree $2t$ using Joint-ZVSS$(t, n, 2t)$. This results in shares b_i, c_i for the uncorrupted players. Erase all secret values aside from b_i, c_i.
   (c) Run the simulator SIM-DL-Key-Gen of the DL-Key-Gen protocol on input β as the "public key". This results in shares a_i for each of the uncorrupted players. Note that all other information has already been erased by the simulator which was called as a sub-routine.
   (d) The simulator knows values $v_i = k_i a_i + b_i, i \in \mathcal{B}$ that should be broadcast by the players controlled by the adversary in Step 1e of the signing protocol. Choose a $2t$-degree polynomial $f_v(z)$ s.t. $f_v(0) = \mu$, $f_v(i) = v_i$ for $i \in \mathcal{B}$. Set $v_i^* = f_v(i)$ for $i \in \mathcal{G}$. Compute $b_i^* = v_i^* - k_i a_i$, $i \in \mathcal{G}$. Erase the secret values b_i for $i \in \mathcal{G}$. Broadcast v_i^* for $i \in \mathcal{G}$.
2. The simulator knows values $s_i = k_i(m + x_i r) + c_i, i \in \mathcal{B}$ that should be broadcast by the players controlled by the adversary in Step 2 of the signing protocol. Choose a $2t$-degree polynomial $f_s(z)$ s.t. $f_s(0) = s$, $f_s(i) = s_i$ for $i \in \mathcal{B}$. Set $s_i^* = f_s(i)$ for $i \in \mathcal{G}$. Compute $c_i^* = s_i^* - k_i(m + x_i r)$, $i \in \mathcal{G}$. Erase the secret values c_i for $i \in \mathcal{G}$. Broadcast s_i^* for $i \in \mathcal{G}$.
3. Erase all the information generated by the signature generation.

Figure 5: SIM-DSS-Sig-Gen - Simulator for the DSS-Sig-Gen Protocol

Proof: 1. Generation of r

(a) The simulator executed Joint-RVSS according to the protocol, thus all the values generated by these executions have the required distribution.

(b) The simulator executed two instances of Joint-ZVSS according to the protocol, thus all the values generated by these executions have the required distribution.

(c) It has been proved that the simulator SIM-DL-Key-Gen outputs the desired distributions for the DL-Key-Gen protocol. Notice also that β is uniformly distributed in G_g (the same as g^a in the real protocol).

(d) Here the simulator broadcasts values v_i^* which were not computed according to the protocol. Yet because these shares have been chosen at random under the condition that they interpolate to a random free term μ and the polynomial interpolated by them matches the shares held by the adversary, the view of the adversary is exactly the same as in the real protocol.

2. Generation of s: The same argument as above applies to the values s_i^*.

The discussion about the internal states presented by the simulator to the adversary when a player is corrupted is identical to the one in the proof of Lemma 1. It is important to notice that the only rewinding happens during the simulation of the DL-Key-Gen subroutine inside this protocol. It should be restated here that once the DL-Key-Gen simulator completes its execution the adversary can now corrupt any of the players, including the "special" player, because now even for that player the simulator has a consistent view. □

Lemma 3 *If DSS is unforgeable under adaptive chosen message attack then* DSS-ts *is an* unforgeable *t-threshold* ts *scheme for $t < n/4$.*

Proof: Assume that DSS-ts is not unforgeable. Then there exists a t-threshold adversary $\mathcal{A}$ s.t. after participating in the initial execution of DL-Key-Gen and then in the repeated execution of DSS-Sig-Gen on messages $m_1, m_2, ...$ of $\mathcal{A}$'s choice, with a non-negligible probability $\mathcal{A}$ outputs a valid DSS (message,signature) pair

$(m, (r, s))$ under the public key (p, q, g, y) produces by DL-Key-Gen. Furthermore, none of the m_i's is equal to m. Using such adversary $\mathcal{A}$, we show how to construct a forger $\mathcal{F}$ against the *regular* DSS scheme. $\mathcal{F}$ is given as input a DSS system (p, q, g), and a random public key y. Additionally, $\mathcal{F}$ can accesses a signature oracle O_{Sig} that provides DSS signatures under the given public key (p, q, g, y).

$\mathcal{F}$ fixes the random coins of $\mathcal{A}$. First $\mathcal{F}$ plays the part of the honest parties in the h-generation protocol on input (p, q, g). Then $\mathcal{F}$ runs an interaction between SIM-DL-Key-Gen and $\mathcal{A}$ on input (p, q, g, h, y). By lemma 1, the simulation ends in expected polynomial time and $\mathcal{A}$ receives a view that is identical to $\mathcal{A}$'s view of a random execution of DL-Key-Gen that outputs y. When $\mathcal{A}$ requests a signature on message m_i, $\mathcal{F}$ submits m_i to O_{Sig} and receives (r_i, s_i), a random DSS signature on m_i. Then $\mathcal{F}$ runs an interaction between SIM-DSS-Sig-Gen and $\mathcal{A}$ on SIM-DSS-Sig-Gen's input $(p, q, g, h, y, m_i, (r_i, s_i))$. By lemma 2, this simulation ends in expected polynomial time and $\mathcal{A}$ receives a view that is identical to $\mathcal{A}$'s view of a random execution of DSS-Sig-Gen on input (p, q, g, h, y, m_i) that outputs (r_i, s_i). Finally, since its views of this simulation are indistinguishable from the real ones, $\mathcal{A}$ outputs a valid DSS signature $(m, (r, s))$ where $m \neq m_i$ for all i. $\mathcal{F}$ outputs this $(m, (r, s))$ which is a successful existential forgery since $\mathcal{F}$ never asked its oracle this message m. $\square$

Lemma 4 *Assuming the computation of discrete logarithms modulo p is hard,* DSS-ts *is a t-threshold robust* ts *scheme for $t < n/4$.*

Proof: Assume that DSS-ts is not t-robust. Then there exists an adversary $\mathcal{A}'$ s.t. with a non-negligible probability, after $\mathcal{A}'$ participates in the initial execution of DL-Key-Gen and then in the repeated execution of DSS-Sig-Gen on messages $m_1, m_2, \ldots$ of $\mathcal{A}$'s choice, the signing servers fail to produce a valid signature (r, s) on some m_i under the public key (p, q, g, y) produced by DL-Key-Gen. Using such adversary $\mathcal{A}'$, we show how to construct an extractor $\mathcal{E}$, a probabilistic polynomial time machine which on input (p, q, g, h) where (p, q, g) is a DSS instance and h is a random element in the subgroup of Z_p^* generated by g, outputs $DLOG_g(h)$. In other words, such $\mathcal{E}$ would compute discrete logarithm on random instances modulo p.

On input (p, q, g, h), $\mathcal{E}$ first simulates the h-picking protocol on inputs (p, q, g) to $\mathcal{A}'$ s.t. this protocol ends with h. Then $\mathcal{E}$ fixes the random coins of $\mathcal{A}'$ and runs an execution of DL-Key-Gen on behalf of the good players. After that, whenever $\mathcal{A}'$ submits messages `'sign` m_i`'` for signature, $\mathcal{E}$ runs DSS-Sig-Gen on behalf of the currently good players. During this execution of both the initial DL-Key-Gen and the later instances of DSS-Sig-Gen, $\mathcal{E}$ turns over players to $\mathcal{A}'$ according to the latter's corruption requests.

We argue that if $\mathcal{A}'$ can disrupt i-th instance of DSS-Sig-Gen from producing a valid signature then $\mathcal{E}$ can "catch" $\mathcal{A}'$ on cheating. More specifically, let y be the public output of the above execution of the DL-Key-Gen protocol. The only way that the i-th instance of DSS-Sig-Gen above can fail to produce a pair (r_i, s_i) s.t. DSS-Ver$((p, q, g, y), m_i, (r_i, s_i)) = \texttt{pass}$ is if $\mathcal{A}'$ manages one of the following:
(1): Some corrupted player P_i cheats in one of the Joint-RVSS (or Joint-ZVSS) protocols that the servers perform in the initial DL-Key-Gen or in the i-th instance of DSS-Sig-Gen. I.e. the shares player P_i distributed pass the verification equation 1 (see Figure 7), but they do not lie on a t-degree polynomial.
(2): Some corrupted player P_i publishes $(A_i^*, B_i^*) \neq (A_i, B_i)$ in Step 2, and then passes the zero-knowledge proof of possession of discrete logarithm, in Steps 2-5, during the initial DL-Key-Gen (Figure 1), or when the code of DL-Key-Gen is invoked in the i-th instance of the DSS-Sig-Gen (step 1c, Figure 4).

On the other hand, whenever $\mathcal{A}'$ manages one of the above, $\mathcal{E}$ can detect it: In case (1), $\mathcal{E}$ sees all the shares of the uncorrupted players, and so it detects if they do not lie on a t-degree polynomial. In case (2), if P_i broadcasts $A_i^* \neq g^{f_i(0)}$, $\mathcal{E}$ can detect it by reconstructing $f_i(z)$. When that happens, $\mathcal{E}$ stops the protocol and computes the discrete logarithm $D = DL_g(h) \bmod p$ in two ways, corresponding to the two ways $\mathcal{A}'$ was cheating:
(1): If in Pedersen VSS, player P gives to the currently uncorrupted players (i.e. to $\mathcal{E}$) shares shares (a_i, b_i) s.t. values $a_i + D * b_i$ lie on a t-degree polynomial (only then the verification equation 1 is met), while a_i's (and b_i's) do not, then $\mathcal{E}$ can solve a system of $t+2$ linear equations of the form $-D * b_i + c_0 + c_1 i + c_2 i^2 + \ldots + c_t i^t = a_i$ with $t + 2$ unknowns: $D, c_0, c_1, \ldots, c_t$. If $\mathcal{A}'$ manages to cheat and hence some $t + 2$ values b_i do not lie on a t-degree polynomial, then there are $t + 2$ equations in this system which are not linearly dependent, and so the system can be solved.
(2): If player P_i submits in Step 2 values $A_i^*, B_i^*, T_i = g^{r_i}, T_i' = h^{r_i'}$ s.t. $A_i^* \neq g^{f_i(0)}$ and $B_i^* \neq h^{f_i'(0)}$, then $\mathcal{E}$ reconstructs D as follows. Denote $f_i(0) = a$, $f_i'(0) = b$, $DLOG_g(A_i^*) = a'$, $DLOG_h(B_i^*) = b'$. $\mathcal{E}$ knows a, b. If $\mathcal{E}$ learns $(a', b') \neq (a, b)$, then since $a + Db = a' + Db'$, $\mathcal{E}$ can compute D. Here is how $\mathcal{E}$ extracts the (a', b')

pair: $\mathcal{E}$ runs the ZK proof steps, i.e. step 3 to 5, and stops. This gives him values $r_i + d * a'$ and $r'_i + d * b'$ for some random challenge d. Then $\mathcal{E}$ rewinds $\mathcal{A}'$ and runs Steps 3 to 5 again, to obtain $r_i + d' * a'$ and $r'_i + d' * b'$ for a new random d'. Since the probability that $d = d'$ is negligible, $\mathcal{E}$ has four equations with four unknowns and can solve for a', b'. □

Threshold DSS with Optimal $t < n/2$ Resilience. The DSS-Sig-Gen protocol of Figure 4 can be modified to achieve an optimally resilient adaptively-secure DSS signature generation protocol. Such protocol, together with the optimally-resilient version of DL-Key-Gen presented at the end of Section 3, leads to an optimally-resilient version of the Threshold DSS Signature Scheme DSS-ts. The modifications of DSS-Sig-Gen we present here are based on the statically-secure optimal-resilient DSS protocol of [GRR98].

The first obstacle to achieve optimal fault-tolerance is the use of error-correcting codes which tolerate only $t < n/3$ faults. This is the same problem as we had in DL-Key-Gen, and we follow here the same recipe that gave us an optimally-resilient DL-Key-Gen, i.e. we use Pedersen's verification equation to sieve-out bad shares.

The other reason for non-optimal fault-tolerance in DSS-Sig-Gen in Figure 4 is that polynomials of degree $2t$ are explicitly reconstructed. This seem to imply that even using Pedersen-based sieving of bad shares (rather than error-correcting) the number of points needed is at least $3t + 1$. There is however a way to avoid direct interpolation of polynomials of degree $2t$ when reconstructing product values (as shown in [BGW88] and in a more efficient way in [GRR98]). For example, during Step 1 in Figure 4 the players have to compute the product $\mu = ka$ where k, a are shared among them. Instead of directly reconstructing the $2t$-degree product polynomial, each player P_i can share $v_i = k_i a_i$ the product of its local share using a polynomial F_i of degree t (a ZK proof is necessary to enforce that P_i really shares the correct value). Notice that μ is a linear combination of $2t + 1$ of the v_i's. Thus μ can be reconstructed as the free term of a linear combination of $2t + 1$ of the sharing polynomials F_i which is a polynomial of degree t.

The only complication left to deal with is how to recover the points of bad players that would not behave properly during the protocol. Say that P_1 is bad. It is possible for the other players to recover k_1 and a_1 in the clear by computing a linear combination of the other shares they hold. To expose P_1 each player P_i shares k_i (similarly for a_i) among the other players using Pedersen's VSS (adding a ZK proof that he is sharing the correct value). This "double-sharing" serves as a sort of back-up of the shares and it is used to expose the shares of players that are clearly bad. Indeed to reconstruct k_1 it is sufficient that at least $t + 1$ good players broadcast the appropriate linear combination of the shares they received from the back-up sharing. The full protocol is described in Figure 6.

5 Further Applications

The techniques introduced in this paper enable us to achieve adaptive security for other threshold public-key systems. Here we sketch our solution to Adaptive Threshold RSA and Adaptive Proactive Solutions.

Adaptive Threshold RSA. We can achieve an adaptively-secure Threshold RSA Signature Generation protocol (but without distributed key generation) with optimal resilience. Furthermore, our protocol runs in a constant number of rounds. We build our solution on the Threshold RSA solution of [Rab98]. The protocol of that paper needs to be modified to use a Pedersen VSS wherever a Feldman VSS is used, the zero-knowledge proofs which appear need to be modified accordingly, and commitments should be of the Pedersen form.

The most interesting change required in this protocol is due to the following: If the current protocol is invoked twice on two different messages, each player gives its partial signature on these messages under its fixed partial key. But as we have seen in Section 2, the simulator is not allowed to rewind back beyond the current invocation of the signature protocol that it is simulating. Clearly, the simulator cannot know the partial keys of all players. If a player for whom he does not know the partial key is broken into during the simulation of the signing of the second message, then the simulator would need to explain both partial signatures of that player, and hence would be forced to rewind beyond the current invocation of the signature protocol.

To avoid this problem, the partial keys need to be changed after each signature generation. This is achieved in a straightforward manner (though it adds a performance penalty to the protocol).

Input: message m to be signed, plus the outputs of (the optimally-resilient) DL-Key-Gen
Public Output: (r, s) the signature of the message m

1. **Generate** $r = g^{k^{-1}} \bmod p \bmod q$

   (a) Generate k. Players execute Joint-RVSS(t). Player P_i sets k_i, k'_i to his share of the secret and the auxiliary secret. For each i the value $\alpha_i = g^{k_i} h^{k'_i} \bmod p$ is public.

   (b) Generate a random value a and $g^a \bmod p$ using (the optimally-resilient) DL-Key-Gen. Player P_i sets a_i, a'_i to his share of the secret a and the auxiliary secret. For each i the value $\beta_i = g^{a_i} h^{a'_i} \bmod p$ is public.

   (c) Back-up k_i and a_i. Each player P_i shares k_i and a_i using Pedersen's VSS. The values $\gamma_i = g^{k_i} h^{\rho_i} \bmod p$ and $\delta_i = g^{a_i} h^{\sigma_i}$ (for some randomizers ρ_i, σ_i) are public. P_i is required to prove in ZK that the value committed to in α_i (resp. β_i) is the same as the value committed to in γ_i (resp. δ_i). This is done using a zero-knowledge proof from [CD98]. This is a 3-move public coin proof and it is performed by all players together computing the challenge as in Steps 3-4 of (the optimally-resilient) DL-Key-Gen. Ignore those that fail this step. At least $t + 1$ good players will pass it.

   (d) Each player P_i, $i = 1, ..., 2t + 1$, shares its value $v_i = k_i a_i \bmod q$ using Pedersen's VSS. The values $\chi_i = g^{k_i a_i} h^{\tau_i}$, $i = 1, ..., 2t + 1$, (for some randomizers τ_i) are public. Each P_i proves in ZK that the value committed in χ_i is the product of the values committed to in α_i and β_i. This is done using a zero-knowledge proof from [CD98]. This is a 3-move public coin proof and it is performed by all players as above.

   (e) If player P_j fails the proof in the above step, his shares a_j, k_j are publicly reconstructed via interpolation of the back-ups of shares distributed in Step 1c. Since each a_j (resp. k_j) can be computed as a linear combination of some $t + 1$ values a_i (resp. k_i) that were properly shared in Step 1c, each player broadcasts the appropriate linear combination of its shares of these a_i's (resp. k_i's), together with their associated randomness (computed as the same linear combination of the associated randomness generated in Step 1c). Bad values are sieved out using the public commitments of Step 1c, a_j and k_j are reconstructed, and v_j is set to $a_j k_j$.

   (f) The value $\mu = ka$ is a linear combination of the values $v_1, ..., v_{2t+1}$. Thus it can be computed interpolating the polynomial of degree t which is a linear combination of the polynomials used in Step 1d to share $v_1, ..., v_{2t+1}$. Each player broadcasts its share (together with its associated randomness) of that t-degree polynomial, which is itself a linear combination of the shares of $v_1, ..., v_{2t+1}$ received in Step 1d. For the v_j's that were exposed in Step 1e, we use the constant sharing polynomial. Bad shares are detected using the public commitments, and μ is reconstructed.

   (g) Player P_i computes locally $\mu^{-1} \bmod q$ and $r \stackrel{\triangle}{=} (g^a)^{\mu^{-1}} \bmod p \bmod q$.

2. **Generate** $s = k(m + xr) \bmod q$

   To reconstruct $s = k(m+xr)$, players perform steps equivalent to Steps 1c-1f above, with the values $m+x_i r$ taking the role of the a_i's, and with s taking the role of μ. The only difference is that in the back-up step equivalent to Step 1c above, the players need only to create the back-ups of the $m + x_i r$ values. In the following steps (equivalent to Steps 1d-1f above), the players reuse the back-ups of the k_i values that were created in Step 1c.

3. Player P_i erases all secret information generated in this signing protocol.

Figure 6: Optimally-resilient, $n \geq 2t + 1$, version of DSS-Sig-Gen protocol

Adaptive Proactive Solutions. Our threshold DSS scheme, as well as other discrete–log based threshold schemes built with our techniques, can be easily proactivized [HJJ+97] by periodic refreshment of the shared secret key. Due to space limitations we simply state here that such refreshment can be achieved in the adaptive model if the players execute a Joint-ZVSS protocol (see Appendix B), and add the generated shares to their current share of the private key x.

References

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds. In *Proc.* 8*th ACM PODC*, pages 201–209. ACM, 1989.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Noncryptographic Fault-Tolerant Distributed Computations. In *Proc.* 20*th STOC*, pages 1–10. ACM, 1988.

[BH92] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Eurocrypt '92*, pages 307–323, 1992. LNCS No. 658.

[BW] E. Berlekamp and L. Welch. Error correction of algebraic block codes. US Patent 4,633,470.

[Can98] R. Canetti. Security and composition of multiparty cryptographic protocols. Available at the Theory of Cryptography Library, http://theory.lcs.mit.edu/∼tcryptol. 1998.

[CCD88] D. Chaum, C. Crepeau, and I. Damgard. Multiparty Unconditionally Secure Protocols. In *Proc.* 20*th STOC*, pages 11–19. ACM, 1988.

[CCH97] R. Canetti, S. Halevi, and A. Herberg. Maintaining Authenticated Communication in the Presence of Break-ins. In *Proc.* 16*th ACM PODC*, ACM, 1997

[CD98] R. Cramer and I. Damgard. Zero-knowledge proof for finite fields arithmetic, or: Can zero-knowledge be for free. In *Crypto '98*, pages 424–441, 1998. LNCS No. 1462.

[CDD+99] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multiparty computations with dishonest minority. In *Eurocrypt '99*, pages 311–326, 1999. LNCS No. 1592.

[CFGN96] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proc.* 28*th STOC*, pages 639–648. ACM, 1996.

[DNS98] C. Dwork, M. Naor, and A. Sahai. Concurrent zero-knowledge. In *Proc.* 30*th STOC*, pages 409–418. ACM, 1998.

[DOW92] W. Diffie, P. van Oorschot, and M. Weiner. Authentication and authenticated key exchange. *Designs, Codes and Cryptography*, (2):107–125, 1992.

[ElG85] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Info. Theory*, IT 31, 1985.

[FMY] Y. Frankel, P. MacKenzie, and M. Yung. Adaptively-secure distributed public-key systems. Personal communication with M.Yung.

[GJKR96] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Eurocrypt '96*, pages 354–371, 1996. Full version: http://www.research.ibm.com/security/DSSthresh.ps.

[GJKR99] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. The (in)security of distributed key generation in dlog-based cryptosystems. In *Eurocrypt '99*, pages 295–310, 1999. LNCS No. 1592.

[GK96] O. Goldreich and H. Krawczyk. On the composition of zero-knowledge proof systems. *SIAM. J. Computing*, 25(1), 1996.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *JCSS*, 28(2):270–299, April 1984.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Computing*, 17(2):281–308, April 1988.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM. J. Computing*, 18(1):186–208, February 1989.

[GRR98] R. Gennaro, M. Rabin, and T Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In *Proc.* 17*th ACM PODC*, pages 101–112. ACM, 1998.

[HJJ+97] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *1997 ACM Conference on Computers and Communication Security*, 1997.

[IS93] T. Itoh and K. Sakurai. On the complexity of constant round zkip of possession of knowledge. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E76-A(1), January 1993.

[KPR98] J. Kilian, E. Petrank, and C. Rackoff. Lower bounds for zero-knowledge on the internet. In *Proc.* 39*th FOCS*, pages 484–492. IEEE, 1998.

[Lan95] S. Langford. Threshold DSS signatures without a trusted party. In *Crypto '95*, pages 397–409, 1995. LNCS No. 963.

[NIST91] National Institute for Standards and Technology. Digital Signature Standard (DSS). Technical Report 169, August 30 1991.

[Ped91a] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Crypto '91*, pages 129–140, 1991. LNCS No. 576.

[Ped91b] T. Pedersen. A threshold cryptosystem without a trusted party. In *Eurocrypt '91*, pages 522–526, 1991. LNCS No. 547.

[Rab98] Tal Rabin. A simplified approach to threshold and proactive RSA. In *Crypto '98*, pages 89–104, 1998. LNCS No. 1462.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[Y82] A. C. Yao, Theory and Applications of Trapdoor functions. In *Proc.* 23*rd FOCS*, pages 80–91. IEEE, 1982.

A Adaptive Security of Threshold Signature Schemes: Definitions

This section describes the model of computation and defines adaptively secure threshold signature schemes. The definition is similar to the unforgeability definition of Gennaro et. al. [GJKR96], with natural adaptation to the adaptive model.

The Model of Computation. There are n servers $P_1 ... P_n$ who implement the distributed "signature service", and an adversary $\mathcal{A}$, all of which can be thought of as probabilistic polynomial time interactive Turing machines. The parties have a security parameter k and random inputs $\vec{r} = r_0, r_1 ... r_n$ (r_0 for the adversary, r_i for server P_i); in addition the adversary has some auxiliary input, z, that represents external, prior information. Note that the servers do not have inputs or auxiliary inputs (see Remark 1 below).

We assume a partially synchronous communication network among the servers, which consists of an authenticated broadcast channel as well as ideally secure point-to-point channels between each two servers. The adversary $\mathcal{A}$ learns the messages sent on the broadcast channel, and can send messages on it. By partial synchrony we mean that the computation proceeds in synchronized rounds, but the communication is only weakly synchronized within the round. More precisely, we model this as follows: In each round, the adversary decides in the adaptive fashion, , based on the informatio known to it so far, which *uncorrupted* parties to activate and which to corrupt. An adversary is t-threshold if it never corrupts more than t servers.

When the adversary corrupts a party P, he learns P's *current internal state*. Furthermore, from this point on, all messages directed to P become known to the adversary, who also sends messages in the name of P. When the adversary activates a uncorrupted party P, it receives the messages sent to it in the previous round and it generates its messages for this round. Once all the uncorrupted parties were activated, the adversary generates the messages to be sent by the corrupted servers, and the next round of computation can begin.[13] This partially synchronous model, where the corrupted parties decide on their messages based on the messages that the uncorrupted parties sent *in the same round*, is sometimes referred to as the rushing adversary model.

Threshold Signature (ts) Schemes and their Security. An n-server ts scheme τ is a triple of protocols (Key-Gen, Sig-Gen, Ver), a distributed key generation protocol, a distributed signature generation protocol, and a verification algorithm (see a detailed description below). Our notion of security of ts schemes is a natural generalization of the classic notion of existential *unforgeability* against chosen message attacks [GMR88]. Namely, we require that the adversary should be unable to come up with a valid signature on a message that was not signed by the servers, even after invoking the servers to sign any sequence of adaptively chosen messages (Definition 1). In addition, we require that the signature scheme be *robust*, i.e. that if the uncorrupted servers participate in the signing protocol then a valid signature should be generated (Definition 2). (We make a simplifying assumption that the uncorrupted servers always agree on which messages are to be signed. See also Remarks 2 and 3 below.) We formalize the three components (Key-Gen, Sig-Gen, Ver) of a ts scheme as follows:[14]

• Key-Gen, a key generation protocol, is run by the n servers. Each server has a security parameter k, random input r_i, and generates a private output s_i and a public output v. Without loss of generality we assume that the public outputs of all the uncorrupted servers are identical and are sent on the broadcast channel.

• Sig-Gen, a signature generation protocol, is run by the n servers. This protocol is invoked by an adversary when he sends a message of a form `'sign` (m)`'` on the broadcast channel. Each server has then a public input m, its private input s_i (presumably, s_i is the servers' private output from a run of the key generation protocol), and its random input r_i. This protocol has only public output, denoted by `'signed` (m, u)`'`, which contains a signature u on message m. Also here we assume that the public outputs of all the uncorrupted servers are identical

[13] For example, take a simple protocol between players P_1, P_2, P_3, which consists of two computation rounds: In the first round each player P_i broadcasts its input bit b_i. In the second, each player outputs the xor of the received bits. Suppose that the inputs of the players are $1, 0, 0$. The 1-threshold adaptive adversary $\mathcal{A}$ could easily force the output of the uncorrupted players in this protocol to be 0. For example, $\mathcal{A}$ could do the following: (1) Activate P_1 and P_2, which causes them to broadcast their input bits 1 and 0; (2) Corrupt P_3 and learn its input bit 0; (3) Since all the uncorrupted parties have been activated, $\mathcal{A}$ can now decide to broadcast $b'_3 = 1$ as the input bit of P_3, and the first round of the computation ends; (4) In the second round, $\mathcal{A}$ activates players P_1 and P_2 again, at which point they both receive bits $1, 0, 1$ which were broadcasted in the previous round, and compute $1 \oplus 0 \oplus 1 = 0$ as their output.

[14] In the description of Sig-Gen the adversary is the only user of the signature service implemented by the ts. Such formalization allows for concise definitions of both unforgeability and robustness of a ts scheme, where we assume that the adversary invokes the signature protocol either (or both) to learn to forge signatures, or to try to disrupt the servers from succesfully executing the protocol.

and are sent on the broadcast channel.

• Ver, a signature verification algorithm, can be run by any party. It takes an input v and a pair (m, u). (Presumably, v is the public output of a the key generation protocol, m is some message, and u is a signature.) The output is `pass/fail`.

An adversarial-run of a ts scheme $\tau = (\mathsf{Key\text{-}Gen}, \mathsf{Sig\text{-}Gen}, \mathsf{Ver})$, with adversary $\mathcal{A}$ (having auxiliary input z) on random input $\vec{r}$, consists of first running protocol $\mathsf{Key\text{-}Gen}(k; \vec{r})$, with the participation of $\mathcal{A}$, and random inputs $\vec{r}$ for the parties. Let $v, \vec{s} = s_1 ... s_n$ denote the public and private outputs from Key-Gen, respectively. Next, each time the adversary submits a message `'sign` (m)`'` on the broadcast channel, the servers run $\mathsf{Sig\text{-}Gen}(v, m; \vec{s}, \vec{r})$,[15] with the participation of $\mathcal{A}$. We stress that $\mathcal{A}$ can invoke many copies of protocol Sig-Gen. However, all the invocations of Sig-Gen are assumed to be sequential, i.e. a new invocation does not start until the previous one is completed. The output of this adversarial run of the threshold signature scheme τ, denoted $\mathsf{out}_{\tau,\mathcal{A}}(k, z, \vec{r})$, is the public output v of Key-Gen, followed by the concatenation of all the messages of the type `'sign` (m)`'` sent on the broadcast channel by the adversary, and the messages of the type `'signed` (m, u)`'` sent on the broadcast channel by the parties and the adversary. Let $\mathsf{out}_{\tau,\mathcal{A}}(k, z)$ denote the distribution of $\mathsf{out}_{\tau,\mathcal{A}}(k, z, \vec{r})$ where $\vec{r}$ is uniformly chosen from its domain. Now we can formalize the notions of unforgeability and robustness of a ts scheme:

Definition 1 (Unforgeability of ts schemes) *A ts scheme $\tau = (\mathsf{Key\text{-}Gen}, \mathsf{Sig\text{-}Gen}, \mathsf{Ver})$ is t-threshold unforgeable if for any t-threshold adversary $\mathcal{A}$, and any auxiliary input z, the random variable $\mathsf{out}_{\tau,\mathcal{A}}(k, z)$ satisfies the following property, except for probability negligible in k: If $\mathsf{out}_{\tau,\mathcal{A}}(k, z)$ contains a message `'signed` (m, u)`'`, such that $\mathsf{Ver}(v, m, u) = \mathtt{pass}$, then it contains a message `'sign` (m)`'`.*
Informally: If the output includes a message `'signed` (m, u)`'` with a valid signature u on a message m, then this message must have been produced by the distributed signature protocol invoked on message m by a message `'sign` (m)`'` that also appears in the output. In other words, the adversary $\mathcal{A}$ cannot produce signatures otherwise but by asking the signing servers to do it. I.e., the adversary cannot forge signatures.

Definition 2 (Robustness of ts schemes) *A ts scheme $\tau = (\mathsf{Key\text{-}Gen}, \mathsf{Sig\text{-}Gen}, \mathsf{Ver})$ is t-threshold robust if for any t-threshold adversary $\mathcal{A}$, and any auxiliary input z, the random variable $\mathsf{out}_{\tau,\mathcal{A}}(k, z)$ satisfies the following property, except for probability negligible in k: If $\mathsf{out}_{\tau,\mathcal{A}}(k, z)$ contains a message `'sign` (m)`'` then it also contains a message `'signed` (m, u)`'` s.t. $\mathsf{Ver}(v, m, u) = \mathtt{pass}$.*
Informally: If the signing servers are asked to sign some message m (i.e. `'sign` (m)`'` appears in the output) then they always produce a valid signature on that message. I.e., the adversary cannot disrupt the signature process.

Definition 3 (Security of ts schemes) *A ts scheme τ is t-threshold secure if it is both t-threshold robust and t-threshold unforgeable.*

Remark 1: On the influence of other protocols run by the servers. Our definition of security postulates that, upon corrupting a server, the adversary sees the current internal state of the server; however, only the state of the current execution of the ts scheme is seen. In particular, the adversary does not see internal data that pertains to other protocols that may be in execution at that server.

This simplification of the model is inconsequential in our model. In particular, it can be seen that the proofs of our schemes remain valid (with immediate modifications) even in the case where the adversary receives some additional "auxiliary input" when corrupting a server.

Remark 2: On weaker communication models. Our definition and protocols are presented in a very convenient communication model, where the parties are provided with point-to-point secure channels, as well as an authenticated broadcast channel. Removing the secrecy and authenticity assumptions from the point-to-point channels may be done by using standard message authentication mechanisms, and adaptively secure encryption schemes (e.g., [BH92, CFGN96], see the last paragraph of Section 2). The authenticated broadcast channel can

[15] Presumably, the servers use a different part of the random vector $\vec{r}$ as their source of randomness for the execution of Key-Gen and each instance of Sig-Gen.

be implemented via Byzantine agreement protocols. That is, whenever a server wants to broadcast a message, an agreement protocol on the value of this message is invoked.

Note that if there is no broadcast channel then defining security of ts schemes becomes considerably more tricky. In particular, it is no longer clear that all servers invoke each copy of Sig-Gen with the same input message. See also Remark 3.

Remark 3: Other definitions of security of ts schemes. A somewhat different approach to defining security of ts schemes is taken in [CCH97]. There, an ideal distributed signature service is formulated; A ts scheme is secure if it "emulates" the ideal process, in some well defined way. That formalization provides a natural way for defining security even when no broadcast channels (nor any authenticated communication channels) exist. In the model studied here (secure channels, broadcast channel), the [CCH97] definition seems equivalent to ours.

B Protocols for Shared Generation of a Secret (Joint-RVSS) and Shared Generation of a Randomizing Polynomial (Joint-ZVSS)

Threshold Parameter: t', the degree of the created secret sharing polynomial
Input: Parameters (p, q, g) and element h generated by g
Public Output: C_{ik} for $i = 1..n$, $k = 0..t'$ (referred to as the "commitment to the polynomial"). Set $QUAL$ of non-disqualified players
Secret Output of P_i: x_i and x'_i the share and the associated random value
$f_i(z), f'_i(z)$ the polynomials used to share z_i
s_{ji}, s'_{ji} for $j = 1..n$ shares received from P_j

1. Each player P_i performs a Pedersen-VSS of a random value z_i as a dealer:
   (a) P_i chooses two random polynomials $f_i(z), f'_i(z)$ over Z_q of degree t':
   $f_i(z) = a_{i0} + a_{i1}z + ... + a_{it'}z^{t'}$ $\quad f'_i(z) = b_{i0} + b_{i1}z + ... + b_{it'}z^{t'}$
   Let $z_i = a_{i0} = f_i(0)$. P_i broadcasts $C_{ik} = g^{a_{ik}}h^{b_{ik}} \bmod p$ for $k = 0, ..., t'$.
   P_i sends shares $s_{ij} = f_i(j), s'_{ij} = f'_i(j) \bmod q$ to each P_j for $j = 1, ..., n$.
   (b) Each P_j verifies the shares received from other players for $i = 1, ..., n$
   $$g^{s_{ij}}h^{s'_{ij}} = \prod_{k=0}^{t'} (C_{ik})^{j^k} \bmod p \quad (1)$$
   If the check fails for an index i, P_j broadcasts a *complaint* against P_i.
   (c) Each player P_i who, as a dealer, received a complaint from player P_j broadcasts the values s_{ij}, s'_{ij} that satisfy Eq. (1).
   (d) Each player builds the set of players $QUAL$ which excludes any player
   - who received more than t complaints in Step 1b, or
   - answered to a complaint in Step 1c with values that violate Eq.(1).
2. The shared random value x is not computed by any party, but it equals $x = \sum_{i \in QUAL} z_i \bmod q$. Each P_i sets his share of the secret to $x_i = \sum_{j \in QUAL} s_{ji} \bmod q$ and the associated random value $x'_i = \sum_{j \in QUAL} s'_{ji} \bmod q$.

Figure 7: Joint-RVSS - Joint Pedersen VSS (also a basis of Joint-ZVSS protocol)

A crucial component of our protocols is a sub-protocol Joint-RVSS (Figure 7) (and its modification, Joint-ZVSS discussed below) that selects a random number x uniformly distributed in Z_q, such that each player holds a share

x_i of x and no other information with respect to x is revealed. The sharing is Shamir-based with polynomial of degree t', where t' is a parameter of the protocol (our protocols DL-Key-Gen and DSS-Sig-Gen evoke Joint-RVSS and Joint-ZVSS with t' equal either to t or $2t$). Protocol Joint-RVSS achieves this outcome with a joint execution of Pedersen's information-theoretically private VSS [Ped91a, Ped91b]. This protocol is fully and concisely presented in [GJKR99]. The security of this protocol in the non-adaptive adversarial model is also analyzed in [GJKR99], and the same analysis applies to the adaptive model treated here.

Joint-RVSS proceeds as follows: Each player in the network selects a random number $z_i \in Z_q$ and shares this secret value with Pedersen's VSS using a polynomial of degree t'. At the end of this stage a set $QUAL$ of players who successfully completed the VSS is fixed. The random number x is set to the sum of the z_i properly shared (i.e. for the indices $i \in QUAL$). Locally, each player can compute his share x_i of x by summing up the shares received by the players in $QUAL$. The public output of the protocol contains also values C_{ik} which represent information-theoretically private commitments to the polynomials used by the players to share z_i.

The distributed signature protocol DSS-Sig-Gen (Figure 4, Section 4) uses a variation of the Joint-RVSS protocol, where "the secret" that each player secret-shares is equal to zero. Such polynomial is used to randomize an already existing secret sharing. We refer to this protocol as Joint-ZVSS, for Joint Zero Verifiable Secret Sharing. It follows the protocol of Joint-RVSS in Figure 7 with the following modifications: In Step (1a), each player chooses $a_{i0} = b_{i0} = 0$, and in Step (1b) each player P_j additionally checks for each i that $C_{i0} = 1 \bmod p$.

C The Digital Signature Standard (DSS)

The Digital Signature Standard (DSS) [NIST91] is a signature scheme based on the El-Gamal [ElG85] and Schnorr's [Sch91] signature schemes, which was adopted as the US standard digital signature algorithm. Following [GJKR96], our description of the DSS protocol uses the notation introduced in [Lan95], which differs from the original presentation of [NIST91] by switching k and k^{-1}. This change allows a clearer presentation of our threshold DSS signature protocols in Section 4.

Key Generation. A DSS key is composed of public information p, q, g, a public key y and a secret key x, where:
1. p is a prime number of length l, where l is a multiple of 64 and $512 \leq l \leq 1024$.
2. q is a 160-bit prime divisor of $p-1$.
3. g is an element of order q in Z_p^*. The triple (p, q, g) is public.
4. x is the secret key of the signer, a random number $1 \leq x < q$.
5. $y = g^x \bmod p$ is the public verification key.

Signature Algorithm. Let M be the message to be signed. The message is first hashed using the hash function SHA-1, let m be the resulting hash value. The signer picks a random number k such that $1 \leq k < q$, calculates $k^{-1} \bmod q$, and sets

$$\begin{aligned} r &= (g^{k^{-1}} \bmod p) \bmod q \\ s &= k(m + xr) \bmod q \end{aligned}$$

The pair (r, s) is a signature of m.

Verification Algorithm. A signature (r, s) of a message M can be publicly verified by first computing the SHA-1 hash m of M and then by checking that $r = (g^{ms^{-1}} y^{rs^{-1}} \bmod p) \bmod q$ where s^{-1} is computed modulo q.