Threshold Homomorphic Encryption in the Universally Composable Cryptographic Library

Peeter Laud[1,2]* and Long Ngo[1]

[1] Tartu University
[2] Cybernetica AS
peeter.laud@ut.ee, ngothanglong@yahoo.com

Abstract. The *universally composable cryptographic library* by Backes, Pfitzmann and Waidner provides Dolev-Yao-like, but cryptographically sound abstractions to common cryptographic primitives like encryptions and signatures. The library has been used to give the correctness proofs of various protocols; while the arguments in such proofs are similar to the ones done with the Dolev-Yao model that has been researched for a couple of decades already, the conclusions that such arguments provide are cryptographically sound.
Various interesting protocols, for example e-voting, make extensive use of primitives that the library currently does not provide. The library can certainly be extended, and in this paper we provide one such extension — we add threshold homomorphic encryption to the universally composable cryptographic library and demonstrate its usefulness by (re)proving the security of a well-known e-voting protocol.

1 Introduction

Cryptographic protocol verification is an error-prone task. A tractable way of doing it usually involves employing some abstraction of cryptographic operations, for example using the Dolev-Yao model [23]. In this model, messages are modeled as terms over a certain algebra, possibly with some cancellation rules, and possible operations are defined over the structure of those terms. This approach makes it simple to use formal methods to analyse the protocol, but the question of soundness of the abstraction has not been satisfactorily solved yet. On the other hand, computational methods can produce computationally sound proofs, but are complex and error-prone.

There exists a sound abstraction of cryptographic operations — the *universally composable cryptographic library* [12, 11, 7] — that has the abstraction level comparable to the Dolev-Yao model. The first version of this library contained signature and public-key encryption schemes. Later, the library has been extended to some more primitives common in the Dolev-Yao model, and shown that it can not unconditionally have some special primitives.

* Supported by Estonian Science Foundation, grant #6944, and European Union through the European Regional Development Fund and the 6th Framework Programme project AEOLUS (FP6-IST-15964)

In this paper we extend the library to have threshold homomorphic encryption. The extension involves adding some new commands to the library, while maintaining its abstraction level. We show that the extended library is still computationally sound. It is suitable for the analysis of important classes of protocols, for example electronic voting, auctions or lotteries. A separate contribution of this paper is also the actual introduction of a possible Dolev-Yao style abstraction for threshold homomorphic encryption.

2 Related Work

The model of *universal composability* alias *reactive simulatability* was proposed by Canetti [20] and by Pfitzmann et al [34, 13]. The model has been used to define sound abstractions of various cryptographic primitives. Among the most celebrated abstractions is the universally composable cryptographic library [12, 11, 7] providing Dolev-Yao-style abstractions for common cryptographic primitives, namely symmetric and asymmetric encryption, signatures, MACs, nonces. The library has been used in the security proofs for some protocols: Needham-Schroeder-Lowe protocol (asymmetric) [6], the Otway-Rees protocol (symmetric) [2], the strengthened Yahalom protocol (real secrecy) [10], a payment system [3]. There have also been approaches in using the library to construct secure systems [30, 5, 1, 35]. It is also known that certain primitives cannot be reasonably abstracted by such a library [14, 8]. Recently, the notion of *Conditional Reactive Simulatability* [4] has been proposed, providing soundness only for a certain class of users and potentially allowing to abstract more primitives.

Threshold homomorphic encryption [33, 22] is one of the most versatile cryptographic primitives, combining the distribution of trust with the ability to combine plaintexts under encryption. An overview of using this primitive in e-voting is given in [32]. Such e-voting protocols have been proved to be universally composable [27]. In these proofs, the security requirement put on the threshold homomorphic encryption primitive has basically been security under chosen plaintext attacks (IND-TCPA) [24, 25]. We are not aware of any attempts to abstract this primitive in the Dolev-Yao-style.

3 UC Cryptographic Library

In the treatment of reactive simulatability [34] the *systems* are modeled as sets of *structures.* A structure *Str* is a collection of probabilistic interactive Turing machines. Each of the machines has a number of input and output *ports*; an input and output port with the same name (which must not repeat) form a secure communication channel between corresponding machines. Authentic or insecure channels are modeled using secure channels. A port in the structure may also be unconnected; a certain subset S of such ports are called the *free ports.* The structure provides the intended service over those ports. The rest of unconnected ports represent the possible weaknesses of the structure; the adversary will connect to those ports. A *configuration* of (Str, S) is a *closed collection* (i.e. no unconnected ports), consisting of the machines in Str, a machine H representing the users of the service (called the *environment* in [20]) and connecting only to the free ports of the structure, and an adversarial machine A. There may also be connection(s) between H and A. The *view* of the user H in some configuration C, denoted $\mathit{view}_C(\mathsf{H})$, is the distribution of the sequence of messages on the ports of H.

Given two structures Str and Str' with the same set S of free ports, we say that Str is *at least as secure as* Str' if for all H and A there exists an adversary S, such that $\mathit{view}_{\mathit{Str}\|\mathsf{H}\|\mathsf{A}}(\mathsf{H}) \approx \mathit{view}_{\mathit{Str}'\|\mathsf{H}\|\mathsf{S}}(\mathsf{H})$, where $\approx$ denotes computational indistinguishability [26]. The simulatability is *black-box* if there exists a single machine Sim, called the *simulator*, such that $\mathsf{Sim}\|\mathsf{A}$ is a suitable choice for S, for all H and A. The *at least as secure as*-relation is lifted to systems in the natural way. The central result of the theory of UC is the *composition theorem*. It states that if we replace a substructure of some structure with something that is at least as secure, then the entire resulting structure is also at least as secure as the original one.

When modeling and analysing systems, one speaks about real and ideal structures. The real structure reflects the distribution of components in the real world, with each participant typically having one or several machines implementing the cryptographic operations and protocol logic. A typical ideal structure consists of a single machine that "obviously" satisfies the security requirements we have put on the system. A well-designed ideal structure also has a simple internal state and does not use hard-to-analyse operations (e.g. random number generation). One has to show that the real structure is at least as secure as the ideal structure. While analysing a system, one may locate the real structures it is using, replace them with the corresponding ideal structures (using the composition theorem) and analyse the resulting ideal system instead.

The *simulatable cryptographic library* [12] is such a (set of) pair(s) of real and ideal structures. The ideal structure quite precisely imitates the Dolev-Yao terms used to abstract the cryptographic messages. The main part of the ideal machine $\mathcal{TH}_n$ for n participants is the database of terms. For each term that has ever been created by one of the participants or the adversary, it records its outermost constructor and immediate subterms. The library also records which parties know which term; if a party knows a term then it has a *handle* to it. These handles are generated as needed and are themselves devoid of information (they are just consecutive integers). Hence all message transmissions have to happen through the library, which has to translate the handles. For a term t, let $t^{\mathsf{hnd}_\mathsf{u}}$ be its handle for the user u; u may be omitted if it is clear from the context. All parties can store "raw" bit-strings in the database (called the payloads) and $\mathsf{retrieve}$ their contents, construct new terms and decompose them. The rules for the possibility of composing and decomposing terms are very similar to the rules in the Dolev-Yao model. The adversary has some extra commands in its disposal, for example, creating garbage terms or invalid ciphertexts. The machine $\mathcal{TH}_n$ has the input port $\mathsf{in}_{\mathsf{u}_i}?$ and the output port $\mathsf{out}_{\mathsf{u}_i}!$ for communicating with the i-th user and the ports $\mathsf{in}_\mathsf{a}?$ and $\mathsf{out}_\mathsf{a}!$ for communicating with the adversary. In the real structure, the users still access the terms through the handles (because the real and the ideal interfaces must be the same), but there is a machine M_i for each participant P_i. Bit-strings are used to represent cryptographic messages; the machines use them to communicate with each other (and with the adversary). The cryptographic operations are implemented using conventional primitives. Each secure channel between two parties is modeled by one, each authentic or insecure channel by two (from the transmitter to the adversary, and from the transmitter or the adversary to the receiver) communication channels.

In the model of asynchronous relative simulatability [34], the machines themselves are in charge of scheduling. The scheduling is channel-based, each channel is scheduled by a certain machine. Whenever a machine finishes its step and stores the newly generated messages in the buffers of channels it has output ports for, it may also *clock* at most one of the channels it schedules. The first message in the buffer of that channel is then delivered to its recipient and this machine is the next to run. If this buffer is empty or if no channel was clocked then the control passes to a designated machine (usually the adversary) called the *master scheduler*. Such clocking mechanism is very versatile and allows one to model both network delays (channel is clocked by the adversary) and API calls (there is a channel in each direction between two machines, clocked by their transmitters and scheduled each time they are written to). The commands from the user H to $\mathcal{TH}_n$ / M_i are made through API calls. The machine $\mathcal{TH}_n$ also communicates with the adversary using API calls. In the real structure, the machine M_i clocks the channels from itself to the adversary, while the channels from the adversary or between two machines are clocked by the adversary.

More details on the asynchronous relative simulatability and the UC cryptographic library can be found in [34, 12], as well as in the full version of this paper [31].

4 Adding Threshold Homomorphic Encryption

4.1 The Cryptographic Primitives

A (t, w)-*threshold* $(\boxdot, \boxplus)$-*homomorphic encryption primitive* is a tuple of algorithms $(\mathcal{K}, \mathcal{E}, \mathcal{D}, \mathcal{Z}, \mathcal{V}, \mathcal{C})$ where the *key generation* algorithm $\mathcal{K}$ returns a new public key pk, secret keys $sk_1, \ldots, sk_w$ and (public) verification keys $vk_1, \ldots, vk_w$ at each invocation; the encryption $\mathcal{E}_{pk}(m, r)$ returns the encryption of the message m under the public key pk with the random coins r; the decryption $\mathcal{D}_{sk_i}(c)$ returns the *i-th decryption share* ds_i of the ciphertext c, the correctness of decryption can be verified by invoking $\mathcal{V}_{vk_i}(ds_i, p_i, c)$ where $p_i = \mathcal{Z}_{sk_i}(c)$; the *share combination* algorithm $\mathcal{C}$ takes any t decryption shares $ds_{i_1}, \ldots, ds_{i_t}$ and combines them into the plaintext m. For any $(pk, sk_1, \ldots, sk_w, vk_1, \ldots, vk_w)$ possibly returned by $\mathcal{K}$, the algorithms must satisfy the following conditions [27]. The cryptosystem of [22] can be used here.

– Correctness: If $c = \mathcal{E}_{pk}(m, r)$ and $ds_i = \mathcal{D}_{sk_i}(c)$ then $\mathcal{C}(ds_{i_1}, \ldots, ds_{i_t}) = m$.

– Homomorphism: Let $c_i = \mathcal{E}_{pk}(m_i, r_i)$. Then $c_1 \boxdot c_2$ is a valid ciphertext corresponding to the plaintext $m_1 \boxplus m_2$.
– Correct decryption: Let $c = \mathcal{E}_{pk}(m, r)$, $ds_i = \mathcal{D}_{sk_i}(c)$ and $p_i = \mathcal{Z}_{sk_i}(c)$. Then $\mathcal{V}_{vk_i}(ds_i, p_i, c) = \mathsf{true}$.
– Simulatability: there exists and algorithm $\mathcal{S}$ taking as inputs any m, c, and $ds_{i_1}, \ldots, ds_{i_{t'-1}}$ ($t' \leq t$) and returning the (simulated) decryption shares for the rest of the authorities, such that any t of the shares will be combined to m and the simulated shares are indistinguishable from the real shares even to someone with the knowledge of $sk_1, \ldots, sk_{i_{t'-1}}$.
– IND-CPA-security, even if the adversary has learned up to $t-1$ secret keys.

To ease the presentation, we will in the following assume that $vk_i = (pk, i)$. I.e. the public key includes the verification keys.

A *non-interactive zero-knowledge (NIZK) proof* is a message, constructed by a party (the prover) that convinces any other party that the prover knows the witness for the membership of a certain bit-string in a certain language, without leaking any other information.

4.2 Ideal Library

We extend the machine $\mathcal{TH}_n$ to accommodate the new primitive. The extension involves introducing new message constructors for the kinds of data created by the new primitive, as well as commands for generating keys, encrypting, and decrypting messages, verifying and combining shares and performing the homomorphic operations. Foreseeing the application of the extended library in the analyses of various protocols, where the participants must show that the plaintext in the ciphertexts they have produced comes from a restricted set, we parameterize the library with a predicate $\mathbf{V}$ over bit-strings, giving their *validity*. The library allows one to encrypt *only the payloads*, because it is far from clear what the $\boxplus$-combination of non-payload terms should be. The extension adds several new commands and term constructors to $\mathcal{TH}_n$.

To initiate the generation of a new key, a party u (or the adversary; all commands available to a party are also available to the adversary) invokes the command $\mathsf{gen\_enc\_thres\_keylist}(a_1, \ldots, a_w)$, where a_i indicates who receives the i-th share of the secret key (its either a user or the adversary with the adversary receiving at most $t-1$ shares). Upon receiving that command, $\mathcal{TH}_n$ adds to the database a new public key pk (constructor thpk, no arguments) and secret key shares sk_i (constructor thsk, arguments pk, i, a_i) for $1 \leq i \leq w$. It sends *to the adversary* the command $\mathsf{keylist\_notify}(pk^{\mathsf{hnd}}, \mathsf{u}, a_1, \ldots, a_w, sk_{i_1}^{\mathsf{hnd}}, \ldots, sk_{i_k}^{\mathsf{hnd}})$ where $sk_{i_1}, \ldots, sk_{i_k}$ are the secret key shares intended for the adversary. As this command abstracts a certain multiparty computation protocol, the adversary controls when a user learns the key shares intended for it. The adversary may later send a command $\mathsf{adv\_learn\_share}(pk^{\mathsf{hnd}}, j)$, causing $\mathcal{TH}_n$ to send $\mathsf{learn\_share}(pk^{\mathsf{hnd}}, a_1, \ldots, a_w, sk_{i_1}^{\mathsf{hnd}}, \ldots, sk_{i_k}^{\mathsf{hnd}})$ the user u_j, where $sk_{i_1}, \ldots, sk_{i_k}$ are intended for it. The adversary can also generate *invalid keys* by invoking the command $\mathsf{adv\_gen\_key}()$. This causes $\mathcal{TH}_n$ to generate just a single new term for a public key (constructor thpk) and return its handle to the adversary.

The encryption is straightforward: a command $\mathsf{encth}(pk^{\mathsf{hnd}}, m^{\mathsf{hnd}})$ causes $\mathcal{TH}_n$ to create a new term c with the constructor thciph and the arguments pk and m. But $\mathcal{TH}_n$ verifies before that pk is a public key, m is a payload, and $\mathbf{V}(m)$ holds. If the verification is unsuccessful, an error is returned. In addition to c, $\mathcal{TH}_n$ also creates a term $p = \mathsf{nizkv}(c)$ embodying the NIZK proof of correctness of the validity of m. The handles of both c and p are returned. The adversary can also generate an invalid ciphertext or proof — the commands $\mathsf{adv\_invenc}(pk^{\mathsf{hnd}}, l)$ and $\mathsf{adv\_invproof}(pk^{\mathsf{hnd}}, l)$ return handles to terms c and p, respectively, where $c = \mathsf{thciph}(pk, l)$ and $p = \mathsf{nizkv}(l)$. Here l is the intended length of the plaintext. The plaintext itself does not have to be present. Similarly to [12], it is possible to find the public key from a ciphertext using the command $\mathsf{keyofth}$, and it is impossible to use the secret key shares for anything else than decryption.

The decryption command is more complex — it is $\mathsf{decth}(sk^{\mathsf{hnd}}, c_1^{\mathsf{hnd}}, p_1^{\mathsf{hnd}}, \ldots, c_k^{\mathsf{hnd}}, p_k^{\mathsf{hnd}})$, where $c_1, \ldots, c_k$ are ciphertexts and $p_1, \ldots, p_k$ are NIZK proofs. $\mathcal{TH}_n$ verifies that all ciphertexts are created with the public key pk, where sk is the term $\mathsf{thsk}(pk, j, \_)$. $\mathcal{TH}_n$ also verifies that $p_i = \mathsf{nizkv}(c_i)$ (for all i). If some p_i was an invalid proof (had only a length argument) then $\mathcal{TH}_n$ sends $\mathsf{adv\_findwit}(c_i^{\mathsf{hnd_a}}, p_i^{\mathsf{hnd_a}})$ to the adversary and expects to receive $\mathsf{adv\_foundwit}(c_i^{\mathsf{hnd_a}}, p_i^{\mathsf{hnd_a}}, m_i^{\mathsf{hnd_a}})$. If m_i is the plaintext of c_i then $\mathcal{TH}_n$ changes p_i into $\mathsf{nizkv}(c_i)$ and accepts it. While constructing the $\mathsf{adv\_foundwit}(\ldots)$-answer, the adversary is allowed to parse the terms and store new payloads, but not communicate with H. If the checks succeed then $\mathcal{TH}_n$ creates a new payload term d whose payload is the $\boxplus$-combination of the plaintexts of $c_1, \ldots, c_k$. It also creates new terms ds (plaintext share; constructor $\mathsf{thshare}$, arguments $d, j, c_1, \ldots, c_k$) and dp (proof of correctness of decryption, constructor $\mathsf{sharepr}$, argument ds) and returns the handles to the last two terms. The adversary can also use the command $\mathsf{adv\_decth}$ to decrypt; it takes the same arguments, except for NIZK proofs of plaintext validity, and returns a plaintext share and proof of correctness of decryption. The adversary can also construct an *invalid share* by invoking $\mathsf{adv\_invshare}(l, j, c_1^{\mathsf{hnd}}, \ldots, c_k^{\mathsf{hnd}})$, where l is the length of the plaintext, j is the position of the plaintext share and $c_1, \ldots, c_k$ are the ciphertexts from whose combination the plaintext share has been apparently obtained. This command verifies that $c_1, \ldots, c_k$ have the same public key, adds a single new term (constructor $\mathsf{thshare}$, arguments $\perp, j, c_1, \ldots, c_k$) to the database and returns the handle to it. An *invalid proof of correctness of decryption* can also be created by the adversary by invoking $\mathsf{adv\_invdp}()$; it creates a new $\mathsf{sharepr}$-term without arguments. A valid proof can be *transformed* by invoking $\mathsf{adv\_transdp}(dp^{\mathsf{hnd}})$; it creates a copy of the term dp and returns a handle to it.

Finally, there is the command to combine plaintext shares: when receiving $\mathsf{combine}(ds_1^{\mathsf{hnd}}, dp_1^{\mathsf{hnd}}, \ldots, ds_t^{\mathsf{hnd}}, dp_t^{\mathsf{hnd}}, pk^{\mathsf{hnd}})$, $\mathcal{TH}_n$ checks that the decryption shares correspond to the same set of ciphertexts, that they are different, and that the public key is the one that was used to create the ciphertexts. If these checks pass then there are two options. If pk was created by the command $\mathsf{gen\_enc\_thres\_keylist}$ then $\mathcal{TH}_n$ checks that all proofs of correctness of decryption point to their respective decryption shares. If all checks pass, then a handle to the payload d referenced by all ds_i is returned. If pk was created by the command $\mathsf{adv\_gen\_key}$ then $\mathcal{TH}_n$ forwards the $\mathsf{combine}$-command to the adversary (translating the handles in the process) and forwards its answer (which must be a handle to a payload, or $\perp$) back to the user.

The adversary can also invoke a command $\mathsf{adv\_parse}(t^{\mathsf{hnd}})$ for any term for which it has a handle. In most cases, $\mathcal{TH}_n$ answers with the type of t, as well as with t's arguments (the subterms are translated into handles). Only if t is a ciphertext and the adversary does not know enough plaintext and key shares to decrypt, is the adversary given no handle to the plaintext, but is given only the length of the plaintext. Similarly, if t is a plaintext share (its arguments are the plaintext and the ciphertexts whose combination is decrypted) and the adversary is unable to find the plaintext (for the same reasons as above), is the plaintext omitted from the answer of $\mathcal{TH}_n$.

Remark. We assume that while the ideal adversary processes an $\mathsf{adv\_findwit}$-command, its behavior is somehow constrained. Such assumptions on the ideal-process adversary are relatively wide-spread, but little-researched. They appeared already in the original report introducing universal composability [19], where the ideal signature functionality $\mathcal{F}_{\mathrm{SIG}}$ assumed the adversary to return a bit-string representing the signature when asked so. The rationale of putting such restrictions on the ideal adversary are twofold. First, they make the ideal functionality more secure, and second, this restricted class of ideal adversaries is large enough to take into account all possible real adversaries — the composition of the simulator and the real adversary will always belong to this restricted class. We will see more examples of such constraints in Sec. 4.3.

4.3 Real (or Hybrid) Library

The real library for n participants consists mainly of n machines $M_1, \ldots, M_n$ where M_i, having the ports $\mathsf{in}_{\mathsf{u}_i}?$ and $\mathsf{out}_{\mathsf{u}_i}!$ handles the cryptographic tasks for the i-th participant. The machines M_i work as in [12], but they also have to be extended to cope with the new commands. Additionally, we will use certain ideal functionalities for some tasks. These functionalities also have universally composable implementations, with the help of the composition theorem we will get the entire implementation of the real library (in the *common reference string* model [21]).

We make use of the NIZK functionality $\mathcal{F}^R_{\mathrm{NIZK}}$ [28], where R is the *witness relation.* It works as follows. On input $\mathsf{prove}(x, w)$ from some party (including the adversary) it first verifies whether $(x, w) \in R$. If not then it ignores the input. Otherwise it sends $\mathsf{proof}(x)$ to the adversary and expects it to return some bit-string π. $\mathcal{F}^R_{\mathrm{NIZK}}$ stores (x, π) and returns π (representing the proof) to the querying party. To verify a proof, a party submits $\mathsf{verify?}(x, \pi)$ to $\mathcal{F}^R_{\mathrm{NIZK}}$. If (x, π) has been stored, it returns "yes". Otherwise $\mathcal{F}^R_{\mathrm{NIZK}}$ sends $\mathsf{witness}(x, \pi)$ to the adversary and expects it to return some witness w. If $(x, w) \in R$ then $\mathcal{F}^R_{\mathrm{NIZK}}$ stores (x, π) and returns "yes", otherwise it returns "no". While answering to the queries from $\mathcal{F}^R_{\mathrm{NIZK}}$, the adversary *is not allowed to change its state or communicate with other machines.* In other words, the adversary will not remember that it has answered those queries. The simulator given in [28] satisfies this property.

Our real structure contains two machines realizing $\mathcal{F}_{\mathrm{NIZK}}$, with different witness relations. $\mathcal{F}^1_{\mathrm{NIZK}}$ is used to give validity proofs of ciphertexts; its witnessing relation is $R_1 = \{((pk, c), (m, r)) \mid c = \mathcal{E}_{\mathrm{pk}}(m, r) \wedge \mathbf{V}(m)\}$. The machine $\mathcal{F}^2_{\mathrm{NIZK}}$ is used to construct the correctness proofs for decryption; its witnessing relation is $R_2 = \{((ds, c, pk, j), p) \mid \mathcal{V}_{(pk,j)}(ds, p, c) = \mathsf{true}\}$. Those machines have connections to and from the machines $M_1, \ldots, M_n$, as well as the adversary. All communication over those connections is through API calls (subroutine-style), i.e. the sender on a channel also clocks that channel.

Our real structure also contains a machine $\mathcal{F}_{\mathrm{KEY}}$ serving as the ideal functionality for distributed key generation. It also has connections to and from the machines $M_1, \ldots, M_n$ and the adversary. The connections from M_i and between $\mathcal{F}_{\mathrm{KEY}}$ and the adversary are clocked subroutine-style. However, as $\mathcal{F}_{\mathrm{KEY}}$ represents a distributed protocol, the connections from it to machines M_i are clocked by the adversary. $\mathcal{F}_{\mathrm{KEY}}$ accepts a single command $\mathsf{keygen}(a_1, \ldots, a_w)$ from one of the machines $M_1, \ldots, M_n$ or the adversary. Here $a_1, \ldots, a_w$ have the same meaning as by $\mathsf{gen\_enc\_thres\_keylist}$. It responds by generating a set of keys pk, $sk_1, \ldots, sk_w$ and sending to the adversary and all parties mentioned among $a_1, \ldots, a_w$ the public key and all secret key shares intended for this party. Protocols implementing $\mathcal{F}_{\mathrm{KEY}}$ are given e.g. in [37].

Recall that the state of the machines M_i mainly consisted of a dictionary that mapped handles of messages to bit-strings; we assume that the type of each message can be uniquely determined from the bit-string representing it. Let us now describe how M_i processes commands from H. The key-generation command $\mathsf{gen\_enc\_thres\_keylist}(a_1, \ldots, a_w)$ is forwarded to $\mathcal{F}_{\mathrm{KEY}}$ as $\mathsf{keygen}(a_1, \ldots, a_w)$. If some answer is received from $\mathcal{F}_{\mathrm{KEY}}$ (recall that this answer is scheduled by the adversary) then the received public key and secret key shares are stored together with new handles generated for them (we assume that each secret key share includes its position). The handles are also sent to the user as arguments of the command $\mathsf{learn\_share}$.

The command $\mathsf{encth}(pk^{\mathsf{hnd}}, m^{\mathsf{hnd}})$ is realized by performing the same checks as the ideal library, generating random coins r, calling $c^* \leftarrow \mathcal{E}_{pk}(m)$, submitting (pk, c^*) together with the witness (m, r) to $\mathcal{F}^1_{\mathrm{NIZK}}$, getting back p^*, generating new handles c^{hnd} and p^{hnd}, and storing $c^{\mathsf{hnd}} \mapsto (c^*, pk)$ and $p^{\mathsf{hnd}} \mapsto (p^*, c^*)$. Finally, M_i returns c^{hnd} and p^{hnd}. Note that the NIZK proof includes the ciphertext. The command $\mathsf{keyofth}$ is straightforward to implement.

Decryption $\mathsf{decth}(sk^{\mathsf{hnd}}, c_1^{\mathsf{hnd}}, p_1^{\mathsf{hnd}}, \ldots, c_k^{\mathsf{hnd}}, p_k^{\mathsf{hnd}})$ is done by checking all the proofs p_i with the help of $\mathcal{F}^1_{\mathrm{NIZK}}$, combining the ciphertexts as $c = c_1 \boxdot \cdots \boxdot c_k$, decrypting c as $ds^* = \mathcal{D}_{sk}(c)$, finding the proof of correctness by $dp_\circ = \mathcal{Z}_{sk}(c)$, turning it into a NIZK proof of correctness by submitting $(ds^*, c, (pk, j))$ with the witness $dp_\circ$ to $\mathcal{F}^2_{\mathrm{NIZK}}$ and getting back dp^* (here j is the position of sk among the secret key shares; M_i has stored it alongside sk), generating new handles ds^{hnd} and dp^{hnd}, and storing $ds^{\mathsf{hnd}} \mapsto (ds^*, j, c_1, \ldots, c_k)$ and $dp^{\mathsf{hnd}} \mapsto (dp^*, ds^*, j, c_1, \ldots, c_k)$. Here j is the position of the secret key share sk (stored together with it). The newly generated handles are returned. Again note that the proof contains its subject plaintext share which in turn contains the ciphertexts it was generated from. Finally $\mathsf{combine}$ (whose argument was a list of handles to plaintext shares, correctness proofs of decryption, and the public key) is implemented by verifying all the proofs with the help of $\mathcal{F}^2_{\mathrm{NIZK}}$ and combining the shares using the algorithm $\mathcal{C}$.

Theorem 1 *The real structure consisting of machines $M_1, \dots, M_n$, $\mathcal{F}^1_{\mathrm{NIZK}}$, $\mathcal{F}^2_{\mathrm{NIZK}}$, $\mathcal{F}_{\mathrm{KEY}}$ is at least as secure (in the black-box sense) as the ideal structure consisting of the machine $\mathcal{TH}_n$.*

5 The Simulator

Theorem 1 is proved by constructing a suitable simulator *Sim*. The main task of the simulator is to translate between the views of the real and the ideal adversary. Whenever a message is received from $\mathcal{TH}_n$, the simulator has to assign a bit-string to it and forward it to the real adversary. Whenever a message is received from the real adversary, the simulator has to parse that message and enter it into $\mathcal{TH}_n$, receiving a handle for it in the process. Additionally, the scheduling decisions have to be translated. On the one hand, the simulator *Sim* has the ports $\mathsf{in_a}!$ and $\mathsf{out_a}?$ to communicate with $\mathcal{TH}_n$ (the simulator also clocks the channel $\mathsf{in_a}$). On the other hand, it has all the ports for the real adversary, such that it can play the machines $M_1, \dots, M_n$, $\mathcal{F}^i_{\mathrm{NIZK}}$ and $\mathcal{F}_{\mathrm{KEY}}$ to it. The simulator can be thought of as containing the copies of those machines, although it is possible to intervene with their normal operation. In principle, all channels between those machines also exist, even though both their input and output ports belong to *Sim*. If the channel is also clocked by *Sim*, then one does not have to consider this channel. But there are also some channels from *Sim* to *Sim* (originally from $\mathcal{F}_{\mathrm{KEY}}$ to M_i) that are scheduled by the adversary.

The full description of the simulator, as well as its correctness proof is given in [31]. Here we will only describe some more interesting aspects of its work. The main part of the state of the simulator is a database, similar to the machines M_i. It stores the handles of the messages (coinciding with the handles assigned to terms by $\mathcal{TH}_n$) together with the bit-string representation of those messages. There may be some additional arguments associated with each entry. The state of the simulator also includes the states of the "embedded" machines $\mathcal{F}^i_{\mathrm{NIZK}}$.

To translate the handles received from $\mathcal{TH}_n$ to bit-strings given to the real adversary, parse the term corresponding to the received handle, generate new keys, ciphertexts, etc. for all terms that the simulator has not seen before, use the saved bit-string representations for terms already seen, and combine everything together using the cryptographic operation corresponding to the constructor of the term. The simulator may have difficulties if $\mathcal{TH}_n$ does not allow it to parse a certain term. If this term was a ciphertext (and the simulator does not have access to sufficiently many secret key shares to decrypt it) then translate it by generating a random ciphertext and let the embedded $\mathcal{F}^1_{\mathrm{NIZK}}$ give a validity proof for it. If the untranslatable term was a decryption share then construct a bit-string corresponding to it by invoking the share simulation algorithm $\mathcal{S}$ (if the simulator can obtain a handle to the plaintext term) or by generating a random bit-string (otherwise). The matching proof of validity is given by the *embedded* $\mathcal{F}^2_{\mathrm{NIZK}}$, whose operation is modified to *not require a witness*.

The translation of bit-strings received from the real adversary to terms entered into $\mathcal{TH}_n$ is similar — parse the bit-string as much as possible, using the information already available to the simulator, enter the subterms into $\mathcal{TH}_n$ and finally use a message constructor operation to create the term corresponding to the entire bit-string (or some other command available to honest users to obtain a handle to an already existing term). If the simulator cannot fully parse the received bit-string, then one of the adversarial commands of $\mathcal{TH}_n$ has to be used to construct a suitable term; the set of adversarial commands given in Sec. 4.2 is sufficient for all cases. A bit-string representing a public key that the simulator has not yet seen is entered into $\mathcal{TH}_n$ by using the command $\mathsf{adv\_gen\_key}$. A ciphertext encrypted with such a key is entered with the help of the command $\mathsf{adv\_invenc}$. This command also returns the handle for a suitable proof of plaintext validity. If another proof for the same ciphertext is received from the real adversary then $\mathsf{adv\_invproof}$ is used to create a handle corresponding to it. The received decryption shares are treated similarly — if the adversary does not have a handle to the necessary secret key share then the commands for creating invalid shares and/or validity proofs are used. Note that the choice between transforming and existing proof (command $\mathsf{adv\_transdp}$) and generating an invalid proof (command $\mathsf{adv\_invdp}$) depends on whether the corresponding decryption share already has a validity proof in the database of $\mathcal{TH}_n$.

Note that by containing a copy of $\mathcal{F}_{\mathrm{KEY}}$, the simulator knows the secret key shares for all key generations initiated by honest participants (through $\mathcal{TH}_n$), as well as by the real adversary, if it chose to use the functionality $\mathcal{F}_{\mathrm{KEY}}$ for it. The commands $\mathsf{adv\_gen\_key}$ and $\mathsf{adv\_invproof}$ are only necessary if the real adversary has generated the keys without any help from the simulator.

6 Example: a Simple e-Voting System

To demonstrate the usefulness of our extension, we construct a simple e-voting system based on it and prove that it satisfies certain security properties. The system runs with n voters and w authorities. The functionality of the i-th voter is implemented by the machine $\mathsf{M^V}_i$ and the functionality of the j-th authority by the machine $\mathsf{M^{AU}}_j$. These machines can be seen as parts of the honest user H of $\mathcal{TH}_{n+w}$. They receive commands (to vote in a particular way, to start tallying) from the rest of the honest user, implement the voting protocol, and use our library to implement the cryptography and networking. In other words, they are the *protocol machines* of [30]. Any number of machines $\mathsf{M^V}_i$ and at most $(t-1)$ of the machines $\mathsf{M^{AU}}_j$ may be under adversarial control (only static corruptions are allowed).

Later we will recall a number of security requirements for voting systems and show that this system (using the ideal library) meets these requirements. By the composition theorem, the security of the e-voting system is still preserved when we replace the ideal library with the real one.

We put an additional condition on the adversary: when interfering with the communication from voters to authorities, it treats all authorities equally. I.e., it may block a voter transmitting its vote to the authorities, but it may not allow the vote to reach some authorities and not reach the others. If the adversary changes the message sent from a voter to the authorities, all adversaries will still receive the same message. This restriction models the bulletin board that is typically used for the voters to publish their (encrypted) votes [27]. We also assume that the user(s) of $\mathsf{M^V}_i$ and $\mathsf{M^{AU}}_j$ make sure that different phases of voting (key distribution, voting, tallying) start and end at the same time for different machines.

Initialization Each machine $\mathsf{M^V}_i$ generates a signing and verification key pair $(k_i^{\mathsf{s}}, k_i^{\mathsf{v}})$ using the command $\mathsf{gen\_sig\_keypair}$ [12] and sends it to all other parties *over the authentic channel*. Some party (or the adversary) invokes $\mathsf{gen\_enc\_thres\_keylist}(\mathsf{AU}_1, \ldots, \mathsf{AU}_w)$. The authorities will learn their respective secret key shares $sk_1, \ldots, sk_w$ and the encryption key pk. The public key is also transmitted to the voters in an authentic manner. This might be realized by having each authority send pk to each voter over an authentic channel and let the voter accept if it has received the same pk at least t times.

Voting Fig. 1 (left) describes the actions of $\mathsf{M^V}_i$ upon receiving the command $\mathsf{vote}(v)$ from the user for the first time (each subsequent time, the command is ignored). For sending messages to multiple receivers, one has to handle the scheduling [34], but we will omit the details here. Fig. 1 (right) describes the actions of $\mathsf{M^{AU}}_j$ upon receiving a vote l^{hnd}, apparently from the voter $\mathsf{M^V}_i$. After $\mathsf{M^{AU}}_j$ has successfully received the vote of $\mathsf{M^V}_i$, it ignores the subsequent attempts to send it.

Tallying Fig. 2 (left) describes the actions of $\mathsf{M^{AU}}_j$ after receiving the command to count the votes and publish a share of the final result. As usual, the final result is presumed to be the $\boxplus$-combination of the votes. Fig. 2 (right) describes how the votes are combined in any machine.

We see that the system we have thus defined can only be used for a single voting. It would be straightforward to modify it for several elections, by adding a *session identifier* to each command. This session identifier must then be bound to the messages the parties send to each other, requiring the authorities to also sign their messages (plaintext shares). The same key can be used for several elections, in contrast to [27]. Such possibility is given by the functionality $\mathcal{F}_{\mathrm{NIZK}}$, which can be implemented so, that the simulator has a trapdoor for extracting witnesses, and does not have to resort to rewinding the user H.

```
$vote^{\mathsf{hnd}} \leftarrow \mathsf{store}(v)$
$(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \leftarrow \mathsf{encth}(pk^{\mathsf{hnd}}, vote^{\mathsf{hnd}})$
$s^{\mathsf{hnd}} \leftarrow \mathsf{sign}(k_i^{\mathsf{s,hnd}}, c^{\mathsf{hnd}})$
$l^{\mathsf{hnd}} \leftarrow \mathsf{list}(s^{\mathsf{hnd}}, p^{\mathsf{hnd}})$
for all $i \in \{1, ..., w\}$ do
    $\mathsf{send\_i}(\mathsf{AU}_i, l^{\mathsf{hnd}})$
end for
```

```
$s^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, 1)$
$p^{\mathsf{hnd}} \leftarrow \mathsf{list\_proj}(l^{\mathsf{hnd}}, 2)$
$c^{\mathsf{hnd}} \leftarrow \mathsf{msg\_of\_sig}(s^{\mathsf{hnd}})$
if $\mathsf{verify}(s^{\mathsf{hnd}}, k_i^{\mathsf{v,hnd}}, c^{\mathsf{hnd}})$
    and $(c^{\mathsf{hnd}}, p^{\mathsf{hnd}}) \notin image(S)$ then
    $S \leftarrow S \cup \{i \mapsto (c^{\mathsf{hnd}}, p^{\mathsf{hnd}})\}$
            /* Initially, $S$ is empty */
end if
```

Fig. 1. Algorithms for sending and receiving a vote

```
$(ds^{\mathsf{hnd}}, dp^{\mathsf{hnd}}) \leftarrow$
        $\mathsf{decth}(sk^{\mathsf{hnd}}, S(1), \ldots, S(n))$
$l^{\mathsf{hnd}} \leftarrow \mathsf{list}(ds^{\mathsf{hnd}}, dp^{\mathsf{hnd}})$
for all $i \in \{1, ..., w\}$ do
    $\mathsf{send\_i}(\mathsf{AU}_i, l^{\mathsf{hnd}})$
end for
for all $i \in \{1, ..., n\}$ do
    $\mathsf{send\_i}(\mathsf{V}_i, l^{\mathsf{hnd}})$
end for
```

```
$num\_shares \leftarrow num\_shares + 1$
        /* Initially, $num\_shares = 0$ */
$C \leftarrow C \cup \{num\_shares \mapsto (ds^{\mathsf{hnd}}, dp^{\mathsf{hnd}})\}$
if $num\_shares \geq t$ then
    for all $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots num\_shares\}$ do
        $res^{\mathsf{hnd}} \leftarrow \mathsf{combine}(C(i_1), \ldots, C(i_t), pk^{\mathsf{hnd}})$
        if $res^{\mathsf{hnd}} \neq \perp$ then
            $res \leftarrow \mathsf{retrieve}(res^{\mathsf{hnd}})$
            Output $res$ to the user and stop
        end if
    end for
end if
```

Fig. 2. Algorithms for tallying and for combining the results

6.1 Security of the e-voting system

Several security properties have been defined for e-voting protocols. Some properties can only be satisfied by policies or voting procedures. We only mention here the security properties in terms of cryptography.

An e-voting protocol should have the following properties [32].

Correctness The voting results must be computed from only legitimate votes.
Privacy Voter's preferences are private.
Coercion-freeness A voter can not later prove that he/she voted in a particular way (Then he can not be forced to vote for something he does not prefer).
Independence A voter should know his vote.

We claim that the e-voting system described above is secure in the above sense. Section 6.2 shows the proof. When the e-voting system uses the real library instead of the abstract one, the properties automatically preserved.

6.2 Proof for the ideal setting

The arguments below are quite similar to those in the Dolev-Yao model.

Correctness The votes that the authority receives from the voters are signed. Hence the adversary could not change their content. Each authority accepts a vote from some $\mathsf{M}^{\mathsf{V}}{}_i$ only once, hence there are no possibilities for replaying the votes. Because of the adversary simulating the bulletin board, the same set of votes reaches each authority. When combining the plaintext shares, the sets of votes must be the same and at least one of the shares must originate from an honest authority. The plaintext shares cannot be interfered by the adversary, as this would invalidate the correctness proofs of decryption.

Privacy The vote privacy can be defined as the secrecy of payloads [9]. To achieve it, the inability of the adversary to get handles to the actual votes is sufficient. But the terms representing the votes are only ever used in the ciphertexts the voters send to the authorities and possibly also in plaintext shares if the adversary chooses to decrypt a vote. But the adversary has corrupted at most $(t-1)$ authorities, hence it cannot combine the shares of a decrypted vote.

After the tallying phase, the adversary learns only the result because all of authorities just give the decryption shares for the correct result.

Coercion-freeness After the voting protocol, a voter has handles to his/her vote, the encrypted vote and the proof of validity of the vote. The only way that the ideal library allows one to verify that the ciphertext represents the vote, is to decrypt the ciphertext. The adversary does not have sufficiently many shares of the decryption key for that.

Independence The library does not offer any means for a party to change the plaintext of a ciphertext without decrypting that ciphertext. Hence an adversarial voter cannot change the vote of an honest voter and present it as its own. An adversarial voter cannot even copy the vote of another voter because algorithm 1 does not allow repetitions.

7 Discussion and Conclusions

We have extended the UC cryptographic library with threshold homomorphic encryption. While extending it, we have made some design choices, the optimality of which can only be decided by using the library in the design and analysis of protocols. In particular, we have chosen to verify the proofs of validity and decryption correctness at the time where the ciphertexts and plaintext shares are actually used. One could imagine the existence of special commands to verify those proofs, and a condition on the user of the library (leading to *conditional reactive simulatability* [4]) to always verify the proofs before decrypting or combining. With the current choice we have avoided introducing the conditions, thereby making the presentation of the library simpler.

As threshold homomorphic encryption is widely applied, this extended library can be used in analysing various protocols. It enables us to achieve computationally sound proofs for a larger class of protocols, including e-voting, in an easier way (by tools or even by hand). As an example, we specify and analyse a simple e-voting protocol in Appendix 6.

Conditional reactive simulatability puts conditions on the *user* of some cryptographic primitive with UC-secure abstraction. In this paper we have shown that it may be equally important to consider restrictions on the possible behavior of the *adversary* trying to attack the *ideal* system. We have seen that the application of the composition theorem may combine those conditions in various ways, sometimes leading to their disappearance. This phenomenon certainly warrants a more thorough investigation.

UC cryptographic library, when combined with conditions put on the user and on the adversary, is one approach possibly leading to machine-assisted verification of security of cryptographic protocols and larger systems. A rather different approach is the *sequence-of-games* approach [29, 16–18, 36] that has seemingly received more attention recently. We believe both approaches have their unique merits and both deserve attention.

References

1. Pedro Adão and Cédric Fournet. Cryptographically sound implementations for communicating processes. In ICALP 2006 (LNCS 4052), pp. 83–94.
2. Michael Backes. A Cryptographically Sound Dolev-Yao Style Security Proof of the Otway-Rees Protocol. In ESORICS 2004 (LNCS 3193), pp. 89–108.
3. Michael Backes and Markus Dürmuth. A cryptographically sound Dolev-Yao style security proof of an electronic payment system. In CSFW 2005, pp. 78–93.
4. Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. *Int. J. Inf. Sec.*, 7(2):155–169, 2008.
5. Michael Backes and Peeter Laud. Computationally sound secrecy proofs by mechanized flow analysis. In ACM CCS 2006, pp. 370–379.
6. Michael Backes and Birgit Pfitzmann. A Cryptographically Sound Security Proof of the Needham-Schroeder-Lowe Public-Key Protocol. In FSTTCS 2003 (LNCS 2914), pp. 1–12.
7. Michael Backes and Birgit Pfitzmann. Symmetric Encryption in a Simulatable Dolev-Yao Style Cryptographic Library. In CSFW 2004, pp. 204–218.
8. Michael Backes and Birgit Pfitzmann. Limits of the cryptographic realization of Dolev-Yao-style XOR. In ESORICS 2005 (LNCS 3679), pp. 178–196.
9. Michael Backes and Birgit Pfitzmann. Relating Symbolic and Cryptographic Secrecy. In IEEE S&P 2005, pp. 171–182.
10. Michael Backes and Birgit Pfitzmann. On the cryptographic key secrecy of the strengthened Yahalom protocol. In SEC 2006 (IFIP 201), pp. 233-245.
11. Michael Backes, Birgit Pfitzmann, and Michael Waidner. Symmetric authentication within a simulatable cryptographic library. In ESORICS 2003 (LNCS 2808), pp. 271–290.
12. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A Universally Composable Cryptographic Library. In ACM CCS 2003, pp. 220–230.
13. Michael Backes, Birgit Pfitzmann, and Michael Waidner. A General Composition Theorem for Secure Reactive Systems. In TCC 2004 (LNCS 2951), pp. 336-354.
14. Michael Backes, Birgit Pfitzmann, and Michael Waidner. Limits of the BRSIM/UC soundness of Dolev-Yao models with hashes. In ESORICS 2006 (LNCS 2189), pp. 404–423.

15. Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A Concrete Security Treatment of Symmetric Encryption. In FOCS 1997, pp. 394–403.
16. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In EUROCRYPT 2006 (LNCS 4004), pp. 409–426.
17. Bruno Blanchet. A computationally sound mechanized prover for security protocols. In IEEE S&P 2006, pp. 140–154.
18. Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In CSF 2007, pp. 97–111.
19. Ran Canetti. A unified framework for analyzing security of protocols. *ECCC*, 8(16), 2001.
20. Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In FOCS 2001, pp. 136–145.
21. Ivan Damgård. Efficient Concurrent Zero-Knowledge in the Auxiliary String Model. In EUROCRYPT 2000 (LNCS 1807), pp. 424–436.
22. Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In PKC 2001 (LNCS 1992), pp. 119–136.
23. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.
24. Pierre-Alain Fouque and David Pointcheval. Threshold Cryptosystems Secure against Chosen-Ciphertext Attacks. In ASIACRYPT 2001 (LNCS 2248), pp. 351–368.
25. Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. Sharing decryption in the context of voting or lotteries. In FC 2000 (LNCS 1962), pp. 90–104.
26. Oded Goldreich. *Foundations of Cryptography. Volume 1 - Basic Tools*. Cambridge University Press, 2001.
27. Jens Groth. Evaluating security of voting schemes in the universal composability framework. In ACNS 2004 (LNCS 3089), pp. 46–60.
28. Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In EUROCRYPT 2006 (LNCS 4004), pp. 339–358.
29. Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In IEEE S&P 2004, pp. 71–85.
30. Peeter Laud. Secrecy Types for a Simulatable Cryptographic Library. In ACM CCS 2005, pp. 26–35.
31. Peeter Laud and Long Ngo. Threshold Homomorphic Encryption in the Universally Composable Cryptographic Library. Cryptology ePrint Archive, Report 2008/367.
32. Helger Lipmaa. Secure electronic voting protocols. In *The Handbook of Information Security*. John Wiley & Sons, 2006.
33. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In EUROCRYPT 1999 (LNCS 1592), pp. 223–238.
34. Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In IEEE S&P 2001, pp. 184–200.
35. Christoph Sprenger, Michael Backes, David A. Basin, Birgit Pfitzmann, and Michael Waidner. Cryptographically sound theorem proving. In CSFW 2006, pp. 153–166.
36. Ilja Tšahhirov and Peeter Laud. Application of dependency graphs to security protocol analysis. In TGC 2007 (LNCS 4912), pp. 294–311.
37. Douglas Wikström. Universally composable DKG with linear number of exponentiations. In SCN 2004 (LNCS 3352), pp. 263–277.