Credential Authenticated Identification and Key Exchange

Jan Camenisch[*] Nathalie Casati[*] Thomas Gross[*] Victor Shoup[†]

February 2, 2010

Abstract

Secure two-party authentication and key exchange are fundamental problems. Traditionally, the parties authenticate each other by means of their identities, using a public-key infrastucture (PKI). However, this is not always feasible or desirable: an appropriate PKI may not be available, or the parties may want to remain anonymous, and not reveal their identities.

To address these needs, we introduce the notions of credential-authenticated identification (CAID) and key exchange (CAKE), where the compatibility of the parties' *credentials* is the criteria for authentication, rather than the parties' *identities* relative to some PKI. We formalize CAID and CAKE in the universal composability (UC) framework, with natural ideal functionalities, and we give practical, modularly designed protocol realizations. We prove all our protocols UC-secure in the adaptive corruption model with erasures, assuming a common reference string (CRS). The proofs are based on standard cryptographic assumptions and do not rely on random oracles.

CAKE includes password-authenticated key exchange (PAKE) as a special case, and we present two new PAKE protocols. The first one is interesting in that it is uses completly different techniques than known practical PAKE protocols, and also achieves UC-security in the adaptive corruption model with erasures; the second one is the first practical PAKE protocol that provides a meaningful form of resilience against server compromise without relying on random oracles.

1 Introduction

Secure two-party authentication and key exchange are fundamental problems. Traditionally, the parties authenticate each other by means of their identities, using a public-key infrastucture (PKI). However, this is not always feasible or desirable: an appropriate PKI may not be available, or the parties may want to remain anonymous, and not reveal their identities.

To address these needs, we introduce the notion of credential-authenticated identification (CAID) and key exchange key exchange (CAKE), where the compatibility of the parties' *credentials* is the criteria for authentication, rather than the parties' *identities* relative to some PKI.

We assume that prior to the protocol, the parties agree upon a *policy*, which specifies the types of credentials they each should hold, along with additional constraints that each credential should satisfy, and (possibly) relationships that should hold *between* the two credentials. The protocol should then determine whether or not the two parties have credentials that satisfy the policy, and in the CAKE case, should generate a session key, which could then be used to implement a secure communication session between the two parties. In any case, neither party should learn anything else about the other party's credentials, other than whether or not they satisfied the policy.

[*]IBM Zurich Research Lab

[†]New York University; work done while visiting IBM Zurich Research Lab; supported by NSF grant CNS-0716690

For example, Alice and Bob may agree on a policy that says that Alice should hold an electronic ID card that says her age is at least 18, and that Bob should hold a valid electronic library card. If Alice then inputs an appropriate ID card and Bob inputs an appropriate library card, the protocol should succeed, and, in the CAKE case, both parties should obtain a session key. However, if, say, Alice tries to run the protocol without an appropriate ID card, the protocol should fail; moreover, Alice should not learn anything at all about Bob's input; in particular, Alice should not even be able to tell whether Bob had a library card or not.

As mentioned above, we may even consider policies that require that certain relationships hold between the two credentials. For example, Alice and Bob may agree upon a policy that says that they both should have national ID cards, and that they should live in the same state.

Both of the two previous examples illustrate that the CAKE problem is closely related to the "secret handshake" problem. In the latter problem, two parties wish to determine if they belong to the same group, so that neither party's status as a group member is revealed to the other, unless both parties belong to the same group. There are many papers on secret handshakes (see [JKT08] for a recent paper, and the references therein). The system setup assumptions and security requirements vary significantly among the papers in the secret handshakes literature, and so we do not attempt a formal comparison of CAKE with secret handshakes. Nevertheless, the two problems share a common motivation, and to the extent that one can view owning a credential as belonging to a group, the two problems are very similar.

We also observe that the CAKE problem essentially includes the PAKE (password-authenticated key exchange) problem as a special case: the credentials are just passwords, and the policy says that the two passwords must be equal.

Our Contributions

So that our results are as general as possible, we work in the Universal Composability (UC) framework [Can05]. We give natural ideal functionalities for CAID and CAKE, and give efficient, modularly designed protocols that realize these functionalities. If the underlying credential system is practical and comes equipped with practical proof-of-ownership protocols (such as the IDEMIX system [CL01]), and if the policies are not too complex, the resulting CAKE protocols are fairly practical. In addition, if the credential system provides extra features such as traceability or revocability, or other mechanisms that mitigate against unwanted "credential sharing", then our protocols inherit these features as well.

All of our protocols are proved UC-secure in the adaptive corruption model, assuming parties can effectively erase internal data. Our protocols require a common reference string (CRS), but otherwise make use of standard cryptographic assumptions, and do not rely on random oracles.

As mentioned above, CAKE includes PAKE, and we also obtain two new practical PAKE protocols. The first is a practical PAKE protocol that is secure in the adaptive corruption model (with erasures); this is not the first such protocol (this was achieved recently by [ACP09], using completely different techniques). The second PAKE protocol is a simple variant of the first, but provides security against server compromise: the protocol is an asymmetric protocol run between a client, who knows the password, and a server, who only stores a function of the password; if the server is compromised, it is still hard to recover the password. Our new protocol is the first fairly practical PAKE protocol (UC-secure or otherwise) that is secure against server compromise and that does not rely on random oracles. Previous practical PAKE protocols that provide security against server compromise (such as [GMR06]) all relied on random oracles (and also were analyzed only in the static corruption model).

Outline of the paper

In §2, we provide some background on the UC framework; in addition, we provide some recommendations for improving some of the low-level mechanics of the UC framework, to address some minor problems with the existing formulation in [Can05] that were uncovered in the course of this work. In any case, our results can be understood independently of these recommendations.

In §3, we introduce ideal functionalities for *strong* CAID and CAKE. These ideal functionalities are stronger than we want, as they can only be realized by protocols that use authenticated channels. Nevertheless, they serve as a useful building block. We also discuss there the types of policies that will be of interest to us here, as we want to restrict our attention to policies that are useful and that admit practical protocols.

In §4, we show how a protocol that realizes the strong CAID or CAKE functionalities can be easily and efficiently transformed into a protocol that realizes the CAID and CAKE functionality. The resulting protocol does not rely on authenticated channels. To this end, we utilize the idea of "split functionalities", introduced in [BCL+05]. Although the idea of using split functionalities for nonstandard authentication mechanisms was briefly mentioned in [BCL+05], it was not pursued there, and no new types of authentication protocols were presented. In this section, we review the basic notions introduced in [BCL+05], adjusting the definitions and results slightly to better meet our needs, and give some new constructions, as well.

In §5 we review definitions of UC zero knowledge (UCZK), and provide some new definitions that will be useful to us. UCZK will be a critical building block in the design of our CAID/CAKE protocols. In this section, we discuss a general language of statements we will want to be able to prove, as well as practical implementations of UCZK protocols for proving such statements. In a companion paper, we plan on fleshing out the details of this general framework, but it should be clear, based on these discussions, that there are, in fact, practical UCZK protocols for all the statements we need to prove in our CAID/CAKE protocols.

In §6, we present practical strong CAID/CAKE protocols for some fairly general policies of interest, and prove their security in the UC-framework, assuming secure channels. Using the split functionalities ideal in §4, these protocols can be transformed into practical CAID/CAKE protocols, which do not assume secure channels.

In §7, we present practical strong CAID/CAKE protocols for the equality relation and an interesting relation related to discrete logarithms. The former gives rise to our first new PAKE protocol, while the latter gives rise to our second new PAKE protocol (which provides resilience against server compromise).

Finally, in §8, we present strong CAID/CAKE protocols that are similar to those in §6, which are more efficient, but work for a more specialized (but still interesting) class of policies.

2 Some UC background

Our corruption model is always *adaptive corruptions with erasures*. We believe that allowing adaptive corruptions is important — there are known examples of protocols that are secure with respect to static corruptions, but trivially insecure if adaptive corruptions are allowed. Allowing erasures is a bit of a compromise: on the one hand, properly implementing secure erasures is difficult — but not impossible; on the other hand, if erasures are not allowed, then it becomes very difficult to obtain truly practical protocols, leading to results that are of theoretical interest only.

To streamline the descriptions of ideal functionalities, we assume the following convention in any two-party ideal functionality:

- the adversary may at any time tell the ideal functionality to abort the protocol for one of the parties — the ideal functionality sends the special message `abort` to that party, and does not communicate any further with that party.

In an actual protocol, an `abort` output would be generated when a "time out" or "error" condition was detected; the aborting party will also erase all internal data, and all future incoming messages will be ignored. While not essential for modeling security, it does allow us to distinguish between detectable and undetectable unfairness in protocols.

We clarify here a number of issues regarding terminology and notation in the UC framework. By a *party* we always mean an *interactive Turing machine (ITM).* A party P is addressed by *party ID (PID)* and *session ID (SID).* So if P has PID P_{pid} and SID P_{sid}, then the PID/SID pair $(P_{\text{pid}}, P_{\text{sid}})$ uniquely identifies the party: no two parties in the system may have the same PID/SID pair.

The convention is that the participants of any single protocol instance share the same SID, and conversely, if two parties share the same SID, then they are regarded as participants in the same protocol.

There are no semantics associated with PIDs, other than their role to distinguish participants in a protocol instance. Some authors (sometimes implicitly) tend to use the term "party" to refer to all ITMs that share a PID. We shall not do this: as stated above, for us, a party is always just a single ITM.

In describing protocols and ideal functionalities, we generally omit SIDs in messages — these can always be assumed to be implicitly defined.

2.1 Notions of security

We recall some basic security notions from [Can05], with some extensions in [CR03] and [BCL$^+$05]. We will not be too formal here.

We say that a protocol Π *realizes* a protocol Π^*, if for every adversary A, there exists an adversary (i.e., simulator) A^*, such that for every environment Z, Z cannot distinguish an attack of A on Π from an attack of A^* on Π^*. Here, Z is allowed to interact directly with the adversary and (via subroutine input/output) with parties (running the code for Π or Π^*) that share the same SID.

If we like, we can remove the restriction that parties must share the same SID, which effectively allows Z to interact with multiple, concurrently running instances of a single protocol in the above experiment. With this relaxation, we say that Π *multi-realizes* Π^*. If these multiple instances of Π access a *common* instance of a setup functionality $\mathcal{G}$, then we say that Π multi-realizes Π^* *with joint access to* $\mathcal{G}$. In applications, $\mathcal{G}$ is typically a common reference string (CRS).

The UC Theorem [Can05] implies that if Π realizes Π^*, then Π multi-realizes Π^*. However, if Π makes use of a setup functionality $\mathcal{G}$, then it does not necessarily follow that Π multi-realizes Π^* with joint access to $\mathcal{G}$: one typically has to analyze the multiple-instance experiment directly.

In the above definitions, if Π^* is the ideal protocol associated with an ideal functionality $\mathcal{F}$, then we simply say that Π (multi-)realizes $\mathcal{F}$ (with joint access to $\mathcal{G}$).

We also have some simple transitivity properties: if Π_1 realizes Π_2, and Π_2 realizes Π_3, then Π_1 realizes Π_3; also, if Π_1 multi-realizes Π_2 with joint access to $\mathcal{G}$, and Π_2 realizes Π_3, then Π_1 multi-realizes Π_3 with joint access to $\mathcal{G}$.

A protocol Π may itself make use of an ideal functionality $\mathcal{F}'$ as a subroutine (where an instance of Π may make use of multiple, independent instances of $\mathcal{F}'$). In this case, we call Π an *$\mathcal{F}'$-hybrid protocol.* We may modify Π by instantiating each instance of $\mathcal{F}'$ with an instance of a protocol Π', and we denote the modified version of Π by $\Pi[\mathcal{F}'/\Pi']$. The UC Theorem implies that if Π' realizes

$\mathcal{F}'$, then $\Pi[\mathcal{F}'/\Pi']$ realizes Π. Also, if Π' multi-realizes $\mathcal{F}'$ with joint access to $\mathcal{G}$, then $\Pi[\mathcal{F}'/\Pi']$ multi-realizes Π with joint access to $\mathcal{G}$.

This last statement is essentially a reformulation of a special case of the JUC Theorem [CR03], but in a form that is more convenient to apply. The JUC Theorem requires a protocol that realizes the multi-session extension of an ideal functionality, where subsession IDs (SSIDs) play the role of SIDs. The problem is that the multi-session functionality is too strong: since an SSID is just an ordinary input, a protocol that realizes the multi-session functionality must keep track of all SSIDs ever uses, in order to detect possible re-use. While one could potentially define a weaker form of multi-session functionality, the notion of multi-realization (introduced, somewhat informally, in [BCL+05], and which can be easily be expressed in the Generalized UC (GUC) framework [CDPW07]) seems a more elegant and direct way of joint access to a CRS or similar setup functionality.

2.2 Conventions regarding SIDs

We shall assume that an SID is structured as a *pathname*:

$$name_0/name_1/\cdots/name_k.$$

These pathnames reflect the subroutine call stack: when an honest party invokes an instance of subprotocol as a separate party, the new party has the same PID of the invoking party, and the SID is extended on the right by one element. Furthermore, we shall assume for two-party protocols, the rightmost element $name_k$, called the *basename*, has the form

$$ext : P_{\rm pid} : Q_{\rm pid} : data,$$

where *ext* is a "local name" used to ensure unique basenames, $P_{\rm pid}$ and $Q_{\rm pid}$ are the PIDs of the participants P and Q, and *data* represents shared public parameters. The ordering of these PIDs can be important in protocols where the two participants play different roles.

These conventions streamline and clarify a number of things. In application of the UC Theorem, we will be interested exclusively in protocols that act as subroutines: they are explicitly invoked by a single caller, who provides all inputs, and who receives all outputs.

The main points here are:

- a subroutine is explicitly invoked by the caller;
- the callee implicitly knows where to write its output.

We can (and will) design protocols that deviate from this simple subroutine structure, although the UC Theorem will not directly apply in these cases.

With these restrictions, it also is convenient to make some restrictions on ideal functionalities: *we shall assume that an ideal functionality only delivers an output to a party that has previously supplied the ideal functionality with an input.*

2.3 Party creation and corruption

Many papers often make the implicit assumption that parties are corrupted PID-wise: if P is corrupted, then any party with the same PID as P is also considered corrupted.[1] This assumption is

[1]In fact, the term "party" seems to be used in a number of inconsistent and confusing ways in the literature — as stated above, we use the term "party" as a synonym for ITM.

somewhat problematic, as it may require ideal functionalities to be aware of global state information. Indeed, there does not appear to be a careful formulation of this assumption anywhere in the literature. However, one typically has to impose some constraint on allowable corruption patterns.

Related to the problem of corrupting parties is the problem of allowing the adversary to create parties. Some rules must also be imposed here. Formally speaking, there is no real reason why the adversary needs to actually create any parties at all, with one very important exception: in the UC framework, ideal functionalities communicate directly with "dummy parties" (really, "stubs"), and the PIDs and SIDs of these dummy parties typically play an important role in formulating security properties (such as in an ideal functionality authenticated channels).

We propose a simple set of rules for party creation and corruption. These rules are more restrictive than those in [Can05]; however, as discussed below, we believe that without some kind of additional restrictions, there are some rather serious problems; we also argue that these restrictions are reasonable in many settings.[2]

Before getting into the rules, recall that there are three basic types of messages in a UC execution experiment: input, output, and network communication messages. Also, let us define some terminology, based on the SID conventions introduced in §2.2. We say that party P' is a *child* of a party P if the PIDs of P and P' are the same, and the SID of P' extends the SID of P on the right by a single component. As usual, we define the *descendent* relation as the transitive closure of the child relation. Finally, we say that the adversary, environment, and ideal functionalities (but not the dummy parties that access them) are *special* parties; all other parties are *ordinary.*

Mechanics of creating or corrupting a party. A party P creates another party P' when P sends an *input* message to P', and P' does not already exist. Whenever is sends an input message, P supplies the PID/SID of P', as well as the name or description of the program to be run by P', should P' not already exist. If P creates P' in this way, then only P is allowed to pass inputs to and receive outputs from P'. Also, when delivering an output, P' does not need to specify the identity of P — this is implicitly defined.

This is the only way a party can be created, with the following exceptions: the environment and adversary are created at the start of execution; in addition, ideal functionalities (but not the dummy parties that access them) are created by other means (see below). Otherwise, our rules (unlike those of [Can05]) do not allow the implicit creation of parties by sending them network communication or subroutine output messages.

An ordinary party is corrupted if it receives as an input a special `corrupt` message. The semantics of party corruption are discussed below.

Restrictions on passing inputs. Since party creation and corruption is done by sending inputs, we impose some syntactic restrictions on who is allowed to send inputs to whom.

Input Rule 1. The environment is allowed to pass inputs to ordinary parties, as long as none of these parties is a descendent of any other. These parties are all running the same programs, and in the single-instance UC experiment, they must have the same SID.

Input Rule 2. An ordinary party P may pass an input to an ordinary party P', provided P' is a child of P.

Input Rule 3. No other inputs may be passed between parties.

With these rules, the subroutine-calling relationship corresponds precisely to the syntactically defined child relationship, with exceptions made for the environment.

[2]Our new rules amount to a change in the "control function" in the language of [Can05].

Corruption behavior. The semantics of party corruption do not have to be built into the framework, but we propose here a simple set of rules that should be sufficient for most settings.

Recall that an ordinary party is corrupted when it receives the special input `corrupt` (special parties are never considered to be corrupt). The party may be either a "dummy" party (which accesses an ideal functionality) or a "real" party. A "dummy" party simply forwards the `corrupt` message to the ideal functionality (which presumably tells the adversary that the party is corrupt, along with other optional information).[3] A "real" party sends a message to the adversary, informing the adversary that he is now corrupt, and giving the adversary the contents of its current internal state.[4] In either case, the corrupted party becomes a simple shell that is completely under the adversary's control: all incoming messages (inputs, incoming network communication, subroutine outputs) are forwarded to the adversary, and the adversary may direct the party to send outgoing messages (outputs, outgoing network communication, subroutine inputs) on its behalf; other than these actions, the corrupted party does nothing else.

Ideal functionalities. With the restrictions presented above, we need special rules for creating and accessing ideal functionalities. With the flexibility of the original rules in the UC framework, there was no need for this.

Ideal functionalities are accessed by dummy parties or by the adversary (but *not* by the environment). If an ideal functionality $\mathcal{F}$ is used as a subroutine, then we require that each instance of $\mathcal{F}$ share the same SID of the dummy parties that access it. If $\mathcal{F}$ is used as a setup functionality that is jointly accessed by different protocol instances, this requirement is dropped.

In either case, some mechanism is required to allow instances of $\mathcal{F}$ to be created dynamically by either dummy parties or the adversary, but in such a way that these instances are guaranteed to be running the correct program for $\mathcal{F}$. To this end, we assume a mechanism to create an ITM running a given program so that the program name is securely encoded as part of the PID of the ITM. With such a mechanism, an instance of $\mathcal{F}$ will always have the appropriate program name securely encoded in its PID.

Discussion. In the current formulation of the basic UC framework in [Can05], not enough restrictions are placed on the creation of parties by the adversary, rendering it difficult to reason about the security of protocols.

For example, consider a protocol that relies on authenticated channels, that is, an $\mathcal{F}_{\text{auth}}$-hybrid protocol (see [Can05, p. 60] for the definition of $\mathcal{F}_{\text{auth}}$). Such a protocol actually accesses $\mathcal{F}_{\text{auth}}$ via dummy parties, and $\mathcal{F}_{\text{auth}}$ does nothing more than verify that the PID and SID of the *dummy* sending party are correct. The real sending party gives its message to the dummy sending party, which is transmitted to the dummy receiving party via $\mathcal{F}_{\text{auth}}$, and finally delivered to the real receiving party. If, however, the adversary can create or corrupt a party with the PID and SID that should belong to the dummy sending party, then $\mathcal{F}_{\text{auth}}$ provides no real guarantees at all.[5] To be useful, there must be a way for the real receiving party to infer that if the real sending party is honest, then the dummy sending party is honest and under control of the real sending party. One could attempt to modify $\mathcal{F}_{\text{auth}}$, so that it checked the name and/or the code of the real sending party, but the problem would just re-assert itself in higher-level ideal functionalities (such as zero knowledge).

[3]In the erasure model, we assume that if a party has previously aborted, then no optional information is sent to the adversary.

[4]In the erasure model, we assume that when a party aborts, all internal state is erased.

[5]Indeed, it seems that this example illustrates that technically speaking, all current theorems that purport to prove the security of protocol that use authenticated channels are false.

While the above criticism may seem rather trivial, we believe that, in fact, the basic UC framework is missing a mechanism for enforcing essential trust assumptions, which we can characterize as follows. Suppose P and Q are peers in some protocol, and that P uses a subroutine P' and Q uses a subroutine Q', where P' and Q' are peers in some subprotocol. Certainly, if P is honest, and P creates P', then it should be the case that P' initially runs the code that P chooses. This is indeed enforced in the current formulation of the basic UC framework. However, that is not enough. We really need additional restrictions — like those given here — that guarantee that only P can create P', and that as long as P remains honest, then so does P'.

Are these conventions too restrictive? We do not believe so.

One consequence of these conventions is that a party may be a subroutine of exactly one other party. In the original UC formulation, no such restriction is made. However, from the point of view of applying the UC Theorem (which is the main point of defining the framework in the first place), one might as well make this restriction: the reason is that without this restriction, the a "subroutine respecting" hypothesis will not be satisfied.

However, there are other types of composition where this restriction may not be convenient. As an example, consider the JUC Theorem [CR03]. In some situations, one can use the multi-realization with joint access formulation stated above in §2.1, and avoid the issue completely. However, if one really wants to use the JUC Theorem as it is stated, the problem is also avoided with a simple change in perspective. The JUC Theorem states that if Π is an $\mathcal{F}'$-hybrid protocol that realizes $\mathcal{F}$, and $\hat{\Pi}'$ realizes the multi-session extension $\widehat{\mathcal{F}}'$ of $\mathcal{F}'$, then $\Pi[[\mathcal{F}'/\hat{\Pi}']]$ realizes $\mathcal{F}$. Here, $\Pi[[\mathcal{F}'/\hat{\Pi}']]$ represents a special composition operator, which translates calls to $\mathcal{F}'$ in an instance of Π with calls to a *common* instance of $\hat{\Pi}'$ with SIDs translated to SSIDs. Now, these calls to $\mathcal{F}'$ may not be made by the original instance of Π, but rather by various subroutines, and the resulting calls to $\hat{\Pi}'$ in $\Pi[[\mathcal{F}'/\hat{\Pi}']]$ would violate our single-caller restriction. The solution is simply to view an instance $\Pi[[\mathcal{F}'/\hat{\Pi}']]$ as a single monolithic party, which internally runs a "virtual OS" in which the subroutines of this party actually run — calls to $\mathcal{F}'$ are intercepted by the "virtual OS", and replaced by appropriate calls to $\hat{\Pi}'$. It is easy to see that the JUC Theorem still holds, and we are still able to adhere to our restrictions, both in the "physical world" in which the instances of $\Pi[[\mathcal{F}'/\hat{\Pi}']]$ execute, but also in the "virtual world" in which instances of Π and its subroutines the subroutines execute. Moreover, we are still able to benefit from the UC Theorem, using it to prove that the hybrid protocol Π realizes $\mathcal{F}$ in a modular way.

We will use this "virtual OS" idea again later, in §4.6.

In addition, one should bear in mind that our naming conventions are simply tools to make the UC Theorem work, and that PIDs and SIDs are artificial names that do not have to correspond to real-world network addresses or protocol stacks. For example, a client may try to run a certain protocol with a server. Almost certainly, the protocol stacks of client and server will not look the same; however, the client and server could, for example, exchange nonces and then launch a protocol instance whose SID is "rooted" at a pathname formed by concatenating the two nonces; any subprotocols then extend the pathnames by means of some simple naming convention.

Some further benefits. A benefit of our restrictions is that it simplifies the management of SIDs. A basic requirement of the UC framework is that every party is uniquely named by its PID and SID. Without restrictions such as those proposed here, this is hard to enforce locally. However, with our restrictions, this becomes easy: a protocol designer just has to ensure that no two subprotocol instances invoked (i.e., created) by it have the same basename.

A further benefit is that with our restrictions, the "subroutine respecting" hypothesis of the UC Theorem is automatically satisfied.

2.4 System parameters

A common reference string, or CRS, is sometimes very useful. Sometimes, however, a different, but related notion is useful: a *system parameter*.

Like a CRS, a system parameter is assumed to be generated by a trusted party, but unlike a system parameter, a CRS is visible to *all* parties, including the environment. A nice way to model this is using some elements of the GUC framework.[6]

In designing a protocol that realizes some ideal functionality, a system parameter is a much better type of setup functionality than a CRS, as the security properties of protocols that use a CRS are not always so clear (e.g., "deniability" — see discussion in [CDPW07]). These problems do not arise with system parameters. Moreover, if a protocol Π realizes an ideal functionality $\mathcal{F}$ using a system parameter, then it is easy to see that Π multi-realizes $\mathcal{F}$ as well — there is no need to separately analyze a multi-instance experiment.

A system parameter can also be used to parameterize an ideal functionality — a CRS cannot be used for this purpose, as that would conflate specification and implementation.

We can distinguish between two types of system parameters: *public coin* and *private coin*. In a public-coin system parameter, even the random bits used to generate the system parameter are visible to the environment (but no one else). In a private-coin system parameter, the random bits used to generate the system parameter remain hidden from all parties. We will later see applications of both types.

2.5 Authenticated channels

We present here an ideal functionality for an authenticated channel. We have tuned this functionality to adhere to our conventions. This ideal functionality is called $\mathcal{F}_{\text{ach}}$.

For an SID is of the form $\mathit{sid} := \mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} :$, where P is the sender and Q is the receiver, and for an adversary A, the ideal functionality $\mathcal{F}_{\text{ach}}$ runs as follows:

1. Wait for both:
   (a) an input message $(\texttt{send}, x)$ from P; then send $(\texttt{send}, x)$ to A;
   (b) an input message $\texttt{ready}$ from Q; then send $\texttt{ready}$ to A.
2. Wait for the message $\texttt{deliver}$ from A; then send the output message $(\texttt{deliver}, x)$ to Q.

Corruption rule: If P is corrupted between Steps 1a and 2, then A is allowed to change the value of x (at any time before Step 2).

NOTES:

1. Like the corresponding functionality in [Can05], this one allows delivery of a single message per session. Multiple sessions should be used to send multiple messages. Alternatively, one could also define a multi-message functionality.

2. Unlike the corresponding functionality in [Can05], the receiver here must explicitly initialize the channel before receiving a message. This conforms to our conventions stated above. In

[6] While we will occasionally use some elements of the GUC framework in this paper, we do not claim that any of the protocols we present are fully GUC secure: to achieve this would require a setup functionality, such as an augmented CRS (ACRS), which to realize would require some form of authentication mechanism that we are unwilling to assume in our setting (we are trying to *build* authentication mechanisms).

[Can05], the sender can essentially spontaneously create a party on the receiving end just by sending it a message. While in some settings such behavior may be convenient or even necessary, it does not seem like a good idea to make this the default behavior. Such behavior makes denial-of-service attacks much easier, since the receiving side may be forced to start executing arbitrarily complex protocols (not just the communication protocol, but protocols "up the stack" which may be triggered by it) that are entirely unrelated to any computation it actually wants to perform.

3. Suppose the SID of the ideal functionality is `path/to/foo/bar`. This means that on the sending side, the party that actually produces the message to be delivered (the *producer*) has SID `path/to/foo`, and on the receiving side, the party that actually consumes the message (the *consumer*) also has SID `path/to/foo`. On the sending side, the producer passes its input to a dummy party with SID `path/to/foo/bar`, which in turn passes this to the ideal functionality, which also has SID `path/to/foo/bar`. On the receiving side, the ideal functionality passes the received message to a dummy party with SID `path/to/foo/bar`, which in turn passes this as an output to the consumer. By our conventions and assumptions, if the producer is not corrupt, then the message received by the consumer was actually sent by the producer.

2.6 Secure channels

Secure channels provide both authentication and secrecy. We present a ideal functionality that is tuned to adhere to our conventions, and to our adaptive corruptions with erasures assumption.

For an SID is of the form $\mathit{sid} := \mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} :$, where P is the sender and Q is the receiver, and for an adversary A, the ideal functionality for secure channels runs as follows:

1. Wait for both:
   (a) an input message $(\texttt{send}, x)$ from P; then send the message `send` to A;
   (b) an input message `ready` from Q; then send the message `ready` to A.
2. Wait for the message `lock` from A.
3. Wait for both:
   (a) a message `done` from A; then send the output message `done` to P.
   (b) a message `deliver` from A; then send the output message $(\texttt{deliver}, x)$ to Q.

Corruption rule: If P is corrupted between Steps 1a and 2, then A is given x and is allowed to change the value of x (at any time before Step 2).

NOTES:

1. As above, this functionality only allows a single message per session to be transmitted.
2. A message can only be delivered to a receiver who has initialized the channel.
3. If Q is corrupted between Steps 2 and 3b, then A is allowed to obtain x by sending a `deliver` message to the ideal functionality, obtaining the response $(\texttt{deliver}, x)$, which is available to A, since Q is now under A's control.

Variable length messages. We have left out one important detail in describing our secure channels functionality, namely, variable length messages. The above functionality is suitable for applications where all messages are bounded by some fixed length; however, if message lengths may vary greatly, one may want a more flexible functionality. There are a couple of different possible design choices.

Variation 1. One is to modify Step 1a, so that the message $(\texttt{send}, \text{len}(x))$ is sent to the adversary, where $\text{len}(x)$ is the length of x. This choice reflects the fact that unless all messages are padded out to a maximum length, some information about the message length will invariably be leaked to the adversary.

While simple and convenient, the problem with this approach is that an implementation may require an extra round of interaction for each message delivered.

Variation 2. To address this inefficiency, we may modify the original ideal functionality in the following way:

1. Wait for both:
   (a) an input message $(\texttt{send}, x)$ from P; then send the message $(\texttt{send}, \text{len}(x))$ to A;
   (b) an input message $(\texttt{ready}, \mathit{maxlen})$ from Q; then send the message $(\texttt{ready}, \mathit{maxlen})$ to A.
2. Wait for the message `lock` from A; verify that $\text{len}(x) \leq \mathit{maxlen}$; if not, halt.
3. Wait for both:
   (a) a message `done` from A; then send the output message `done` to P.
   (b) a message `deliver` from A; then send the output message $(\texttt{deliver}, x)$ to Q.

The corruption rule remains the same.

We call this ideal functionality $\mathcal{F}_{\text{sch}}$.

Here, the receiver specifies the maximum length message he is prepared to accept. This functionality reflects the fact that most of the time, the receiver knows the general "size and shape" of the message it is expecting, and so no additional interaction is required. In the cases where this information is not known in advance, the sender can transmit the length information to the receiver ahead of time on an authenticated channel. In fact, this implementation realizes Variation 1 above (at essentially the same cost).

3 Ideal functionalities for strong CAID and CAKE

In this section, we present ideal functionalities for *strong* CAID and CAKE. These ideal functionalities are stronger than we want, as they can only be realized by protocols that use authenticated channels. However, in the next section, we discuss how to we can very easily modify such protocols to obtain protocols that realize the desired CAID/CAKE functionalities (which will be defined in terms of strong CAID/CAKE).

We start with strong CAID.

At a high level, the ideal functionality for strong CAID, denoted $\mathcal{F}^*_{\text{caid}}$, works as follows. We have two parties, P and Q. P and Q agree (somehow) on a binary relation R, which consists of a set of pairs (s, t). Then P and Q submit values to the ideal functionality: P submits a value s and Q a value t. The ideal functionality then checks if $(s, t) \in R$; if so, it sends P and Q the value 1, and otherwise the value 0. The relation R represents the "policy", discussed in §1.

The above description is lacking in details: some essential, and others not. We now describe some detailed variants of the above general idea.

We assume that party P has PID P_{pid} and SID P_{sid}. Likewise, we assume that party Q has PID Q_{pid} and SID Q_{sid}.

3.1 Ideal Functionality $\mathcal{F}^*_{\text{caid}}$

We assume that the SIDs of the two parties are of the form $\mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} : \langle R \rangle$, where $\langle R \rangle$ is a description of the relation R. In principle, any efficiently computable family of relations is allowable, but specific realizations may implement only relations from some specific family of relations. It will convenient to assume that the special symbol $\perp$ has the following semantics: for all s, t, neither $(\perp, t)$ nor $(s, \perp)$ are in R.

An instance of $\mathcal{F}^*_{\text{caid}}$ with SID $\mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} : \langle R \rangle$ runs as follows.

1. Wait for both:
   (a) an input message $(\texttt{left-input}, s)$ from P; then send `left-input` to A;
   (b) an input message $(\texttt{right-input}, t)$ from Q; then send `right-input` to A.
2. Wait for a message `lock` from A; then set
$$\mathit{res} := \begin{cases} 1 & \text{if } (s,t) \in R; \\ 0 & \text{if } (s,t) \notin R. \end{cases}$$
3. Wait for both:
   (a) a message `deliver-left` from A; then send the output message $(\texttt{return}, \mathit{res})$ to P;
   (b) a message `deliver-right` from A, then send the output message $(\texttt{return}, \mathit{res})$ to Q.

Corruption rules:

- If P (resp., Q) is corrupted between Steps 1a (resp., 1b) and 2, then A is given s (resp., t), and is allowed to change the value of s (resp., t) at any time before Step 2.
- If P (resp., Q) is corrupted between Steps 2 and Steps 3a (resp., 3b), then A is given s (resp., t).

Note that the inclusion of $P_{\text{pid}}, Q_{\text{pid}}$ in the SID serves to break symmetry, and establish P as the "left" party and Q as the "right" party.

The above ideal functionality captures the inherent "unfairness" in any such protocol: if one party is corrupt, they may learn that the relation holds, while the other may not. However, such unfairness is at least detectable: since we do not conflate `abort` with a result of 0, if any party is being treated unfairly, this will at least be detected by an `abort` message.

One could consider a weaker notion of security, in which 0 and `abort` were represented by the same value. While this may allow for more efficient protocols, such protocols may allow "undetectable unfairness". With our present formulation, a result of `abort` may indicate an unfair run of the protocol (or it may just indicate that there are network problems).

3.2 Ideal Functionality $\mathcal{F}^*_{\text{pcaid}}$

The functionality $\mathcal{F}^*_{\text{caid}}$ does not provide as much privacy as one might like; in particular, if P and Q are honest, then A still learns the relation R.[7]

To prevent this from happening, we would like to allow P and Q to agree in advance on a description $\langle R \rangle$ of a relation R, by some unspecified means, and then supply $\langle R \rangle$ as a private, shared input. Unfortunately, in the UC framework, there is no convenient way to directly express the notion of a "private shared input". However, we can achieve a similar effect by having one party send $\langle R \rangle$ to the other party (over a secure channel); the second party can either accept the first party's proposal, or not.

For our new functionality, $\mathcal{F}^*_{\text{pcaid}}$ (for strong *private* CAID), the SID is of the form $\mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} :$. Note that $\langle R \rangle$ is no longer a part of the SID.

An instance of $\mathcal{F}^*_{\text{pcaid}}$ with SID $\mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} :$ runs as follows:

0.1 Wait for both:

   (a) an input message ($\texttt{send-relation}, \langle R \rangle$) from P; then send the message ($\texttt{send-relation}, \ell(\langle R \rangle)$) to A;

   (b) an input message `ready` from Q; then send the message `ready` to A.

0.2 Wait for the message `lock-relation` from A.

0.3 Wait for a message `deliver-relation` from A; then send the output message ($\texttt{deliver-relation}, \langle R \rangle$) to Q.

Steps 1–3 are as in $\mathcal{F}^*_{\text{caid}}$, using the relation R described by $\langle R \rangle$. In addition to the corruption rules for $\mathcal{F}^*_{\text{caid}}$, we have:

Additional corruption rules:

- If P is corrupted between Steps 0.1a and 0.2, then A is given $\langle R \rangle$ and is allowed to change the value of $\langle R \rangle$ at any time before Step 0.2.
- If P (resp., Q) is corrupted between Steps 0.2 (resp., 0.3) and 3a (resp., 3b), then A is given $\langle R \rangle$.

In the above, the function ℓ is an "information leakage" function that is used to model the fact that in some circumstances, some information about the general "size and shape" of $\langle R \rangle$ will be revealed to an eavesdropping adversary.

3.3 From authentication to key exchange

Both $\mathcal{F}^*_{\text{caid}}$ and $\mathcal{F}^*_{\text{pcaid}}$ may be extended to provide key exchange in addition to authentication modifying Step 2 as follows:

2. Wait for a message ($\texttt{lock}, K_{\text{adv}}$) from A; then set

$$\mathit{res} := \begin{cases} (1, K) & \text{if } (s,t) \in R, \\ 0 & \text{if } (s,t) \notin R, \end{cases}$$

   where the key K is determined as follows:

[7]PIDs and SIDs are considered public information. Recall that we assume that SIDs are transmitted implicitly as a part of every message. For extra privacy, "real world" users should choose unlinkable pseudonyms as their PIDs, and (to the extent possible) use some type of mechanism to prevent traffic analysis.

- if either P or Q are currently corrupted, set $K := K_{\text{adv}}$;
- otherwise, generate K at random (according to some prescribed distribution).

Corruption rules are unchanged. We call the resulting functionalities $\mathcal{F}^*_{\text{cake}}$ and $\mathcal{F}^*_{\text{pcake}}$, respectively.

3.4 Some relations of interest

One type of relation that is of particular interest is a simple *product relation*, where $R = S \times T$.

For example, we may have $S = \{s : (x, s) \in E\}$, for a given x and a fixed relation E. Here, s might be an "anonymous credential" issued by some authority whose public key is x; the relation E would assert that s is a valid credential relative to x, possibly satisfying some other constraints as well.

A well-known example of an anonymous credential system of this type is the IDEMIX system [CL01]. This system comes with efficient zero-knowledge protocols for proofs of possession of credentials that we will be able to exploit. IDEMIX may also be equipped with mechanisms for identity escrow, revocation, etc., which automatically enhances the functionality of any strong CAID/CAKE protocol.

Similarly, we may have $T = \{t : (y, t) \in F\}$, for a given y and fixed relation F. In this case the description $\langle R \rangle$ of R is the pair (x, y).

Two generalizations of potential interest are as follows.

First, suppose we have binary relations $R_1, \ldots, R_n$. We can define their *vectored union* as the binary relation

$$R = \{((s_1, \ldots, s_n), (t_1, \ldots, t_n)) : (s_j, t_j) \in R_j \text{ for some } i = 1 \mathinner{.\,.} n\}.$$

For example, each relation R_j may represent a pair of "compatible" credentials, and the protocol should succeed if the two parties hold one such pair between them. Or more simply, the two parties may agree on a list of "clubs", and then determine if there is any one club to which they both belong.

Second, we might consider the intersection of a product relation with a *partial equality relation*:

$$\{(s, t) : \sigma(s) = \tau(t)\},$$

where σ and τ are appropriate functions. Such relations can usefully model the "secret handshake" scenario, where $\sigma(s)$ and $\tau(t)$ perhaps represent "group names". A special case of this, of course, is the equality relation. A CAKE protocol for equality is essentially a PAKE protocol — this is discussed in §7.

One might even combine the above, considering vectored unions of such intersections.

The reason for singling out these types of relations is that they are of potential practical interest, and admit efficient protocols.

4 Bootstrapping an authentication protocol

We shall presently give efficient protocols that realize strong CAID/CAKE functionalities for various relations of interest. All of these protocols work assuming secure channels. Of course, this is not interesting by itself, since we really want to use these protocols to establish secure channels in a setting without any existing authentication mechanism.

Without at least authenticated channels, it is impossible to realize strong CAID/CAKE. The solution is to weaken the notion of security, using the idea of "split functionalities", introduced in [BCL$^+$05]. Our definitions of the CAID/CAKE functionalities are simply the split versions of the strong CAID/CAKE functionalities.

Although the idea of using split functionalities for nonstandard authentication mechanisms was briefly mentioned in [BCL$^+$05], it was not pursued there, and no new types of authentication protocols were presented. In this section, we review the basic notions introduced in [BCL$^+$05], adjusting the definitions and results slightly to better meet our needs, and give some new constructions, as well.

4.1 Details: split functionalities

We give a slight reformulation of the definitions and results in [BCL$^+$05]: we focus on the two-party case, and we also make a few small syntactic changes that will allow us to apply the results in a more convenient way.

The basic idea is the same as in [BCL$^+$05]. If $\mathcal{F}$ is a two-party ideal functionality involving two parties, P and Q, then the "split functionality" $\mathsf{s}\mathcal{F}$ works roughly as follows. Before any computation begins, the adversary partitions the set $\{P, Q\}$ into *authentication sets*: in the two-party case, the authentication sets are either $\{P\}$ and $\{Q\}$, or the single authentication set $\{P, Q\}$. The parties within an authentication set access a common instance of $\mathcal{F}$, while parties in different authentication sets access independent instances of $\mathcal{F}$. This is achieved by "mangling" SIDs appropriately: each authentication set is assigned a unique "channel ID" *chid*, which is used to "mangle" the SIDs of the instances of $\mathcal{F}$. Thus, the most damage an adversary can do is to make P and Q run two *independent* instances of $\mathcal{F}$.

As we shall see, one can transform any protocol Π that realizes $\mathcal{F}$, where Π relies on authenticated and/or secure channels, into a protocol $\mathsf{s}\Pi$ that realizes $\mathsf{s}\mathcal{F}$, where $\mathsf{s}\Pi$ relies on neither authenticated nor secure channels. Moreover, $\mathsf{s}\Pi$ is almost as efficient as Π. This result was first proved in [BCL$^+$05]; however, we give a more efficient transformation — based on Diffie-Hellman key exchange — that is better suited to the two-party case.

Our CAID/CAKE functionalities are simply defined as the split versions of the strong CAID/CAKE functionalities: $\mathsf{s}\mathcal{F}^*_{\text{caid}}$, $\mathsf{s}\mathcal{F}^*_{\text{cake}}$, $\mathsf{s}\mathcal{F}^*_{\text{pcaid}}$, and $\mathsf{s}\mathcal{F}^*_{\text{pcake}}$. Protocols for these functionalities may be obtained by applying the split transformation to the protocols for the corresponding strong functionalities.

4.2 General split functionalities

Now we give the general split functionality in more detail. Let $\mathcal{F}$ be an ideal functionality for a two party protocol. As in §2.2, we assume that the SID for $\mathcal{F}$ is of the form $\mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} : \mathit{data}$, and that $\mathcal{F}$ never generates an output for a party before receiving an input from that party.[8]

The split functionality $\mathsf{s}\mathcal{F}$ has an SID $s := \mathit{parent}/\mathit{ext} : P_{\text{pid}} : Q_{\text{pid}} : \mathit{data}$ of the same form as $\mathcal{F}$, and for an adversary A runs as follows.

- Upon receiving a message `init` from a party $X \in \{P, Q\}$: record $(\texttt{init}, X_{\text{pid}})$, send $(\texttt{init}, X_{\text{pid}})$ to A.
- Upon receiving a message $(\texttt{authorize}, X_{\text{pid}}, \mathcal{H}, \mathit{chid})$ from A, such that

[8] [BCL$^+$05] are a bit fuzzy on this point — take a look at Step 3 of the Computation part of their $\mathsf{s}\mathcal{F}$ functionality ...what happens if P_i has not been initialized?

1. $X_{\rm pid}$ is the PID of some $X \in \{P, Q\}$;
2. $\{X_{\rm pid}\} \subseteq \mathcal{H} \subseteq \{P_{\rm pid}, Q_{\rm pid}\}$;
3. $(\texttt{init}, X_{\rm pid})$ has been recorded;
4. no tuple $(\texttt{authorize}, X_{\rm pid}, \ldots)$ has been recorded; and,
5. if a tuple $(\texttt{authorize}, X'_{\rm pid}, \mathcal{H}', \mathit{chid}')$ has been recorded, then either (a) $\mathcal{H}' = \mathcal{H}$ and $\mathit{chid}' = \mathit{chid}$ or (b) $\mathcal{H}' \cap \mathcal{H} = \emptyset$ and $\mathit{chid}' \neq \mathit{chid}$

do the following:

1. if no tuple of the form $(\texttt{authorize}, \cdot, \mathcal{H}, \mathit{chid})$ has already been recorded, then initialize a "virtual" instance of $\mathcal{F}$ with SID
$$\mathit{sid}_{\mathcal{H}} := \mathit{chid}/\mathit{sid};$$
we denote this instance $\mathcal{F}_{\mathcal{H}}$ and define $\mathit{chid}_{\mathcal{H}} := \mathit{chid}$; in addition, for each $Y \in \{P, Q\}$, if $Y_{\rm pid} \notin \mathcal{H}$ or Y is corrupt, then notify $\mathcal{F}_{\mathcal{H}}$ that the party with PID $Y_{\rm pid}$ and SID $\mathit{sid}_{\mathcal{H}}$ is corrupt, and forward to A the response of $\mathcal{F}_{\mathcal{H}}$ to this notification;
2. record the tuple $(\texttt{authorize}, X_{\rm pid}, \mathcal{H}, \mathit{chid})$;
3. send the output message $(\texttt{authorize}, \mathit{chid})$ to X.

- Upon receiving a message $(\texttt{input}, v)$ from $X \in \{P, Q\}$, such that a tuple $(\texttt{authorize}, X_{\rm pid}, \mathcal{H}, \mathit{chid})$ has been recorded: send the message v to $\mathcal{F}_{\mathcal{H}}$, as if coming as an input from the party with PID $X_{\rm pid}$ and SID $\mathit{sid}_{\mathcal{H}}$.

- Upon receiving a message $(\texttt{input}, X_{\rm pid}, \mathcal{H}, v)$ from A, such that

  1. $X_{\rm pid}$ is the PID of some $X \in \{P, Q\}$;
  2. a (uniquely determined) instance $\mathcal{F}_{\mathcal{H}}$ with $X_{\rm pid} \in \mathcal{H}$ has been initialized; and,
  3. $X_{\rm pid} \notin \mathcal{H}$,

  send the message v to $\mathcal{F}_{\mathcal{H}}$, as if coming as an input from the party with PID $X_{\rm pid}$ and SID $\mathit{sid}_{\mathcal{H}}$.

- Whenever an instance $\mathcal{F}_{\mathcal{H}}$ delivers an output v to a party with PID $X_{\rm pid}$, where $X_{\rm pid}$ is the PID of some $X \in \{P, Q\}$, do the following: if $X_{\rm pid} \in \mathcal{H}$, then send the output message $(\texttt{output}, v)$ to X, else send the output message $(\texttt{output}, X_{\rm pid}, v)$ to A.

- Upon receiving notification that a party $X \in \{P, Q\}$ is corrupted, such that a (uniquely determined) instance $\mathcal{F}_{\mathcal{H}}$ with $X_{\rm pid} \in \mathcal{H}$ has been initialized: notify $\mathcal{F}_{\mathcal{H}}$ that the party with PID $X_{\rm pid}$ and SID $\mathit{sid}_{\mathcal{H}}$ is corrupted, and forward to A the response of $\mathcal{F}_{\mathcal{H}}$ to this notification.

We have a slightly different formulation of split functionalities than in [BCL+05], but the differences are mainly syntactic — our method of mangling the SIDs fits nicely in to our set of conventions on SIDs. In addition, in [BCL+05], a party is allowed to send an input as long as its authentication set is defined, whereas we require that a party wait for its explicit authorization notification before proceeding. This seems to avoid some potential confusion.

4.3 A multi-session secure channels functionality

The next goal is to present a "multi-session extension" of our ideal functionality for secure channels. One approach would be to use the definition in [CR03]. However, a direct application of that definition would be unworkable, for two reasons: first, it would require that any implementation keep track of all subsession IDs that were ever used; second, the multi-session extension applies to all possible parties, whereas, we can really only deal with the same two parties in all subsessions. So for these reasons, we present our own multi-session extension, which we denote $\mathcal{F}_{\text{msc}}$. It is quite tedious, and not very enlightening. Note that in addition to secure channels, it also provides for channels that only provide authentication. Also, for secure channels, the ideal functionality is based on the ideal functionality $\mathcal{F}_{\text{sch}}$ in §2.6 that deals with variable length messages.

For an SID $sid := parent/ext : P_{\text{pid}} : Q_{\text{pid}} :$, we define a *valid subsession ID* to be a tuple of the form $(tag, P_{\text{pid}}, Q_{\text{pid}})$ or $(tag, Q_{\text{pid}}, P_{\text{pid}})$, where *tag* is an arbitrary bit string. For an adversary A, the ideal functionality $\mathcal{F}_{\text{msc}}$ then runs as follows:

Init: Initialize a counter *cnt* to 0, and sets *Ready*, *Send*, *Done*, *Deliver*, *AuthReady*, and *AuthSend* to $\emptyset$.

- Upon receiving an input message $(\texttt{ready}, tag, X_{\text{pid}}, Y_{\text{pid}}, maxlen)$ from a party $Y \in \{P, Q\}$, such that $(tag, X_{\text{pid}}, Y_{\text{pid}})$ is a valid subsession ID and Y_{pid} is the PID of Y, do the following: (1) increment *cnt*, (2) add the tuple $(cnt, tag, X_{\text{pid}}, Y_{\text{pid}}, maxlen)$ to *Ready*, and (3) send the message $(\texttt{ready}, tag, X_{\text{pid}}, Y_{\text{pid}}, maxlen)$ to A.

- Upon receiving an input message $(\texttt{send}, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ from a party $X \in \{P, Q\}$, such that $(tag, X_{\text{pid}}, Y_{\text{pid}})$ is a valid subsession ID and X_{pid} is the PID of X, do the following: (1) increment *cnt*, (2) add the tuple $(cnt, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ to *Send*, and (3) send the message $(\texttt{send}, tag, X_{\text{pid}}, Y_{\text{pid}}, \text{len}(v))$ to A.

- Upon receiving a message $(\texttt{lock}, i, j)$ from A such that for some $tag, X_{\text{pid}}, Y_{\text{pid}}, v, maxlen$, the set *Send* contains the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$, the set *Ready* contains the tuple $(j, tag, X_{\text{pid}}, Y_{\text{pid}}, maxlen)$, and $\text{len}(v) \le maxlen$, do the following:

  1. remove the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ from *Send*;
  2. remove the tuple $(j, tag, X_{\text{pid}}, Y_{\text{pid}}, maxlen)$ from *Ready*;
  3. add the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ to both *Done* and *Deliver*.

- Upon receiving a message $(\texttt{done}, i)$ from A such that for some $tag, X_{\text{pid}}, Y_{\text{pid}}, v$, the set *Done* contains the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$, do the following:

  1. remove the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ from *Done*;
  2. if $X \in \{P, Q\}$ is the party whose PID is X_{pid}, send the message $(\texttt{done}, tag)$ to X.

- Upon receiving a message $(\texttt{deliver}, i)$ from A such that for some $tag, X_{\text{pid}}, Y_{\text{pid}}, v$, the set *Deliver* contains the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$, do the following:

  1. remove the tuple $(i, tag, X_{\text{pid}}, Y_{\text{pid}}, v)$ from *Deliver*;
  2. if $Y \in \{P, Q\}$ is the party whose PID is Y_{pid}, send the message $(\texttt{deliver}, tag, v)$ to Y.

- Upon receiving an input message $(\texttt{auth-ready}, tag, X_{\text{pid}}, Y_{\text{pid}})$ from a party $Y \in \{P, Q\}$, such that $(tag, X_{\text{pid}}, Y_{\text{pid}})$ is a valid subsession ID and Y_{pid} is the PID of Y, do the following: (1) increment *cnt*, (2) add the tuple $(cnt, tag, X_{\text{pid}}, Y_{\text{pid}})$ to *AuthReady*, and (3) send the message $(\texttt{auth-ready}, tag, X_{\text{pid}}, Y_{\text{pid}})$ to A.

- Upon receiving an input message $(\texttt{auth-send}, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$ from a party $X \in \{P, Q\}$, such that $(\mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}})$ is a valid subsession ID and X_{pid} is the PID of X, do the following: (1) increment *cnt*, (2) add the tuple $(\mathit{cnt}, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$ to *AuthSend*, and (3) send the message $(\texttt{auth-send}, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$ to A.

- Upon receiving a message $(\texttt{auth-deliver}, i, j)$ from A such that for some $\mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v$, the set *AuthSend* contains the tuple $(i, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$ and the set *AuthReady* contains the tuple $(j, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}})$, do the following:

  1. remove the tuple $(i, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$ from *AuthSend*;
  2. remove the tuple $(j, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}})$ from *AuthReady*;
  3. if $Y \in \{P, Q\}$ is the party with PID Y_{pid}, send the output message $(\texttt{auth-deliver}, \mathit{tag}, v)$ to Y.

- Upon receiving a corruption notice for $X \in \{P, Q\}$, then if X_{pid} is the PID of X, for each tuple in *Send* of the form $(i, \mathit{tag}, X_{\mathrm{pid}}, Y_{\mathrm{pid}}, v)$, A is given this tuple.

NOTES:

1. The adversary is free to effectively change the messages that are in transit when a sender is corrupted — this is done by simply adding the modified messages to the *Send* and *AuthSend* sets, and ignoring the originals.

4.4 Split key exchange

We now discuss a simple, low-level primitive: split key exchange. Let $\mathcal{K}$ be a key set. The ideal functionality $\mathcal{F}_{\mathrm{ske}}$ (parameterized by $\mathcal{K}$) has an SID of the form $\mathit{parent}/\mathit{ext} : P_{\mathrm{pid}} : Q_{\mathrm{pid}} :$, and for an adversary A, runs as follows:

- Upon receiving a message $\texttt{init}$ from a party $X \in \{P, Q\}$: record $(\texttt{init}, X_{\mathrm{pid}})$, send $(\texttt{init}, X_{\mathrm{pid}})$ to A.

- Upon receiving a message $(\texttt{authorize}, X_{\mathrm{pid}}, \mathcal{H}, \mathit{chid}, K)$ from A, such that

  1. X_{pid} is the PID of some $X \in \{P, Q\}$,
  2. $\{X_{\mathrm{pid}}\} \subseteq \mathcal{H} \subseteq \{P_{\mathrm{pid}}, Q_{\mathrm{pid}}\}$,
  3. $K \in \mathcal{K}$,
  4. $(\texttt{init}, X_{\mathrm{pid}})$ has been recorded,
  5. no tuple $(\texttt{authorize}, X_{\mathrm{pid}}, \ldots)$ has been recorded, and
  6. if a tuple $(\texttt{authorize}, X'_{\mathrm{pid}}, \mathcal{H}', \mathit{chid}', K')$ has been recorded, then either (a) $\mathcal{H}' = \mathcal{H}$ and $\mathit{chid}' = \mathit{chid}$ or (b) $\mathcal{H}' \cap \mathcal{H} = \emptyset$ and $\mathit{chid}' \neq \mathit{chid}$,

  do the following:

  1. record the tuple $(\texttt{authorize}, X_{\mathrm{pid}}, \mathcal{H}, \mathit{chid}, K)$;
  2. if $K_{\mathcal{H}}$ is not yet defined, then define it as follows:
     - if $\mathcal{H} = \{X_{\mathrm{pid}}\}$, then $K_{\mathcal{H}} \leftarrow K$, else $K_{\mathcal{H}} \leftarrow_{\mathrm{R}} \mathcal{K}$;
  3. send the output message $(\texttt{key}, \mathit{chid}, K_{\mathcal{H}})$ to X.

We now present a simple protocol, Π_{ske}, that realizes the functionality $\mathcal{F}_{\text{ske}}$, under the decisional Diffie-Hellman (DDH) assumption. Assume a group $\mathbb{G}$ of prime order q generated by $g \in \mathbb{G}$ where the DDH holds. The description of $\mathbb{G}$, q, and g is viewed here as a system parameter. We also assume a PRG that maps a random $w \in \mathbb{G}$ to a pair of keys $(K, K_{\text{auth}}) \in \mathcal{K} \times \mathcal{K}_{\text{auth}}$, where $\mathcal{K}_{\text{auth}}$ is some large set.

For two parties P and Q with SID $sid := parent/ext : P_{\text{pid}} : Q_{\text{pid}} :$, protocol Π_{ske} runs as follows. The roles played by P and Q are asymmetric. The protocol for P runs as follows:

1. P waits for an input `init`; then it computes $x \leftarrow_{\text{R}} \mathbb{Z}_q$, $u \leftarrow g^x$, and sends u to Q.

2. P waits for $v \in \mathbb{G}$ from Q; then it computes $w \leftarrow v^x$, derives keys K, K_{auth} from w using the PRG, sets $chid \leftarrow \langle\, u, v \,\rangle$, sends the key K_{auth} to Q (after erasing all internal state other than $chid$ and K).

3. P waits for a continuation signal, and then outputs and outputs $(\texttt{key}, chid, K)$ (after erasing all internal state).

Note that in the UC framework, a party is allowed to only send one message at a time; therefore, P first sends a message to Q (via the adversary, of course), and then waits for a continuation signal (provided by the adversary) before delivering its own output.

The protocol for Q runs as follows:

1. Q waits for an input `init`; then it then does nothing, except to notify the network (i.e., adversary) that it is ready.

2. Q waits for $u \in \mathbb{G}$ from P; then it computes $y \leftarrow_{\text{R}} \mathbb{Z}_q$, $v \leftarrow g^y$, $w \leftarrow u^y$, derives keys K, K_{auth} from w, erases y, w, sets $chid \leftarrow \langle\, u, v \,\rangle$, and sends v to P.

3. Q waits for $K'_{\text{auth}} \in \mathcal{K}_{\text{auth}}$ from P; then it tests if $K_{\text{auth}} = K'_{\text{auth}}$; if so, it outputs $(\texttt{key}, chid, K)$ (after erasing all internal state).

Theorem 1 *Assuming the DDH for $\mathbb{G}$, an appropriate PRG, and assuming the set $\mathcal{K}_{\text{auth}}$ is large, protocol Π_{ske} realizes the ideal functionality $\mathcal{F}_{\text{ske}}$.*

Proof. Our main task is to design a simulator A^*, given an adversary A. The basic structure of A^* is that it interacts with the ideal functionality $\mathcal{F}_{\text{ske}}$, and runs an instance of A internally.

Init: Initialize a boolean variable $safe \leftarrow \texttt{false}$.

P1: When A^* receives the message $(\texttt{init}, P_{\text{pid}})$ from $\mathcal{F}_{\text{ske}}$, where P is currently honest and Step P1 of the simulation has not already been executed, A^* computes $x \leftarrow_{\text{R}} \mathbb{Z}_q$, $u \leftarrow g^x$, and gives u to A as if it were a message from P to Q.

P2: When A requests that a message $v' \in \mathbb{G}$ be delivered to P, where P is currently honest and Step P1, but not Step P2, of the simulation has already been executed, A^* does the following:

- if Step Q2 of the simulation has been executed, and $u = u'$ and $v = v'$ (i.e., P and Q have had "matching conversations"), and Q is currently honest, then:
  - set $safe \leftarrow \texttt{true}$;
  - choose a random value $\widehat{K}_{\text{auth}} \in \mathcal{K}_{\text{auth}}$ and give this to A, as if it were a message from P to Q.

else:

- derive keys $\overline{K}, \overline{K}_{\text{auth}}$ from $(v')^x$ using the PRG
- give $\overline{K}_{\text{auth}}$ to A, as if it were a message from P to Q.

P3: When A delivers a continuation signal to P, where P is currently honest and Steps P1 and P2, but not Step P3, of the simulation have already been executed, A^* does the following:

- if *safe* then:
  - send the message $(\texttt{authorize}, P_{\text{pid}}, \{P_{\text{pid}}, Q_{\text{pid}}\}, \langle u, v \rangle, K_{\text{def}})$ to $\mathcal{F}_{\text{ske}}$, where $K_{\text{def}} \in \mathcal{K}$ is some default key value (its value is not important);

  else:
  - send the message $(\texttt{authorize}, P_{\text{pid}}, \{P_{\text{pid}}\}, \langle u, v' \rangle, \overline{K})$ to $\mathcal{F}_{\text{ske}}$, where $\overline{K}$ is as computed in Step P2.

Q1: When A^* receives the message $(\texttt{init}, Q_{\text{pid}})$ from $\mathcal{F}_{\text{ske}}$, where Q is currently honest and Step Q1 of the simulation has not already been executed, A^* informs A that Q is ready.

Q2: When A requests that a message $u' \in \mathbb{G}$ be delivered to Q, where Q is currently honest and Step Q1, but not Step Q2, of the simulation has already been executed, A^* computes $y \leftarrow_{\text{R}} \mathbb{Z}_q$, $v \leftarrow g^y$, and gives v to A as if it were a message from Q to P.

Q3: When A requests that a message K'_{auth} be delivered to Q, where Q is currently honest and Steps Q1 and Q2, but not Step Q3, of the simulation have been executed, do the following:

- if *safe* then:
  - verify that K'_{auth} matches the random value of $\widehat{K}_{\text{auth}}$ generated in Step P5 of the simulation; if so, send the message $(\texttt{authorize}, Q_{\text{pid}}, \{P_{\text{pid}}, Q_{\text{pid}}\}, \langle u, v \rangle, K_{\text{def}})$ to $\mathcal{F}_{\text{ske}}$;

  else:
  - derive keys K, K_{auth} from $(u')^y$ using the PRG
  - if $K'_{\text{auth}} = K_{\text{auth}}$ then
    - ∗ send the message $(\texttt{authorize}, Q_{\text{pid}}, \{Q_{\text{pid}}\}, \langle u', v \rangle, K)$ to $\mathcal{F}_{\text{ske}}$.

Pc: Upon receiving a corruption notice for P:

- if the corruption occurs between the execution of Steps P1 and P2 of the simulation, then A^* gives the message x to A, representing the internal state of P;
- if the corruption occurs between the execution of Steps P2 and P3 of the simulation, then A^* gives to A the value K, representing the internal state of P, computed as follows:
  - if *safe* then:
    - ∗ send the message $(\texttt{authorize}, P_{\text{pid}}, \{P_{\text{pid}}, Q_{\text{pid}}\}, \langle u, v \rangle, K_{\text{def}})$ to $\mathcal{F}_{\text{ske}}$, obtaining the response $(\texttt{key}, \textit{chid}, \widehat{K})$ (which is intended for P, but who is now under the control of A^*);
    - ∗ $K := \widehat{K}$;

    else:
    - ∗ $K := \overline{K}$, where $\overline{K}$ is as computed in Step P2 of the simulation.

Qc: Upon receiving a corruption notice for Q:

- if the corruption occurs between the execution of Steps Q2 and Q3 of the simulation, then A^* gives to A the value (K, K_{auth}), representing the internal state of Q, computed as follows:
  - if *safe* then:
    - send the message $(\texttt{authorize}, Q_{\text{pid}}, \{P_{\text{pid}}, Q_{\text{pid}}\}, \langle\, u, v\,\rangle, K_{\text{def}})$ to $\mathcal{F}_{\text{ske}}$, obtaining the response $(\texttt{key}, \mathit{chid}, \widehat{K})$ (which is intended for Q, but who is now under the control of A^*);
    - set $K := \widehat{K}$;
    - set $K_{\text{auth}} := \widehat{K}_{\text{auth}}$, where $\widehat{K}_{\text{auth}}$ is as computed in Step P2 of the simulation;

    else:
    - derive (K, K_{auth}) from $(u')^y$ using the PRG.

That completes the description of A^*. We now argue that A^* is a good simulator.

First, we claim that with overwhelming probability, the "authorize" messages that A^* sends to $\mathcal{F}_{\text{ske}}$ are "consistent", in the sense that either both parties are assigned singleton authentication sets with distinct channel IDs, or both parties are assigned doubleton authentication sets with the same channel ID.

Observe that by the logic of A^*, if P and Q are assigned doubleton authentication sets, then their channel IDs are both equal to $\langle\, u, v\,\rangle$.

Now consider a particular execution of A^* in which both parties are assigned the same channel ID. We want to show that with overwhelming probability, both are assigned doubleton authentication sets. By the logic of A^*, the common channel ID must be $\langle\, u, v\,\rangle$. Now consider what happened when Step P2 was executed.

- Unless a very unlikely event occurred, we may assume that Step Q2 of the simulation has been executed, and $u = u'$ and $v = v'$.

- Further, we claim that unless a very unlikely event occurred, Q has not been authorized yet: suppose to the contrary that it were; this would mean that Step Q3 was already executed, and at that earlier time, A was able to guess the key K_{auth} derived from g^{xy}; but this is a very unlikely event, under the DDH and PRG assumptions, and the assumption that $\mathcal{K}_{\text{auth}}$ is large.

- Since Q has not yet been authorized, but eventually will become authorized, it follows that Q is not currently corrupt and Step Q3 has not yet been executed.

It follows that in Step P2, unless a very unlikely event has occurred, A^* will set $\mathit{safe} \leftarrow \texttt{true}$; later, when Step P3 (or possibly Step Pc) is eventually executed, A^* will find $\mathit{safe} = \texttt{true}$, and so assign P a doubleton authentication set. Similarly, when Step Q3 (or possibly Step Qc) is eventually executed, A^* will find $\mathit{safe} = \texttt{true}$, and so assign Q a doubleton authentication set.

That proves the claim that the authorization messages are consistent. The other thing that one has to show is that the substitution of the real keys K, K_{auth} by random keys is indistinguishable. This follows easily from the DDH and PRG assumptions. The one subtlety to highlight is the fact that this substitution on Q's side only occurs when P has already erased x — this ensures that if P is corrupted after Q outputs the random substitutions, both x and y have been erased, as otherwise, the simulator would be "caught in the act". □

4.5 Realizing split multi-session secure channels

Our goal now is to realize the split version $\mathtt{s}\mathcal{F}_{\text{msc}}$ of the multi-session secure channels functionality $\mathcal{F}_{\text{msc}}$ presented in §4.3. This will be done with an $\mathcal{F}_{\text{ske}}$-hybrid protocol Π_{smsc}, where $\mathcal{F}_{\text{ske}}$ is the split key exchange functionality discussed in §4.4.

At a high-level, Π_{smsc}, works as follows:

1. Wait for an input message `init`; then send the message `init` to $\mathcal{F}_{\text{ske}}$.
2. Wait for a message $(\mathtt{key}, \mathit{chid}, K)$ from $\mathcal{F}_{\text{ske}}$; then:
   (a) derive subkeys required to implement bidirectional secure channels, erasing the key K; these channels will be implemented using a variant of Beaver and Haber's technique [BH92], as roughly described below;
   (b) generate the output message $(\mathtt{authorize}, \mathit{chid})$.
3. Now use the keys derived in the previous step to process the secure channels logic.

More details. We describe at a greater level of detail the logic in implementing the authenticated and secure channels, using the shared key K. There are really two sets of channels: one from P to Q, and one from Q to P. The protocol derives from K separate sets of subkeys for each direction, erasing K, and then processes each direction in the same way, but independently. Thus, we describe the logic generically in terms of a "sender" and "receiver".

The implementation of authenticated channels is easy: sender and receiver share a long-term MAC key, and this is used to authenticate messages.

The implementation of secure channels is a bit more involved. The sender and receiver start out with shared keys consisting of a long-term MAC key and a PRG seed. From this seed, sender and receiver generate (in a coordinated way) subsession keys used for individual subsessions, and update the seed value as they go — for the security proof to work, it is essential that the old seed values are erased. Each subsession key has a corresponding index, corresponding to its position in the pseudo-random sequence.

Each time a "ready" input is received by the receiver, the receiver generates a new subsession key, updates its seed, and sends a "control" message to the sender (authenticated using the long-term MAC key) that consists of (i) the subsession ID, (ii) the value of *maxlen*, and (iii) the index of the subsession key associated with that session. In addition, before sending this control message, the receiver expands the subsession key, using a PRG, to obtain a one-time pad of length *maxlen* and a one-time MAC key, erases the subsession key, and adds a record to a database that consists of the subsession ID, the subsession key index, and the associated one-time pad and one-time MAC key.

Each time a sender receives a "send" input, he adds a record to a database of pending data messages, consisting of the subsession ID and the data message itself. Also, each time the sender receives a control message from the receiver (with a valid MAC), this control message is added to a database of control messages. Whenever the sender finds a pending data message that is consistent with a control message (i.e., the subsession IDs match and the length of the message is no more than the *maxlen* value in the control message), the sender does the following: he deletes the corresponding records from both databases, and expands his seed sufficiently many times to obtain the subsession key of appropriate index (retaining the intermediate subsession keys for possible future use); he then expands the subsession key, and uses these one-time keys to encrypt and MAC the message, erases the message, subsession key and corresponding one-time keys, and sends a "data" message to the receiver consisting of (i) the subsession key index, (ii) the ciphertext, and (iii) the one-time MAC tag. He also sends a "done" message up the local protocol stack. An implementation may choose to send the "data" and "done" messages in either order. Upon receiving this "data" message, the receiver retrieves and deletes the record with matching subsession key index from its own database, verifies the one-time MAC, decrypts the ciphertext, and sends the decrypted message up the protocol stack.

Implementation note. Note that while this implementation requires a good deal of bookkeeping, as well as the additional "control" messages, most of these potential inefficiencies can be avoided in protocols that run in lock step: neither sender nor receiver will be more than one message ahead or behind the other, and so the amount of data stored will be minimal; moreover, most "control" messages can be piggybacked on "data" messages, avoiding extra interaction.

One can prove the following:

Theorem 2 *The $\mathcal{F}_{\mathrm{ske}}$-hybrid protocol Π_{smsc} realizes the ideal functionality $\mathsf{s}\mathcal{F}_{\mathrm{msc}}$, assuming a secure PRG and secure MAC.*

We only sketch the main idea of the proof. When P and Q are assigned to different authentication sets, the protocol essentially provides no security guarantees, so there is not much to say in that case. The interesting case is when P and Q are assigned to the same authentication set. In designing a simulator for this case, the "lock" event for a given subsession occurs when the sender has requested the message to be sent, and the corresponding control message has arrived at the sender. By using one-time pads in this way, it is easy to simulate adaptive corruptions: the "data" message is simulated by just sending a random bit string as a ciphertext; if the sender is subsequently corrupted, all relevant internal state has been erased, and so nothing is leaked; if the receiver is corrupted while the "data" message is in transit, the simulator asks for the message, and then computes a simulated one-time pad by XORing the message and the fake ciphertext.

4.6 Realizing general split functionalities

Let $\mathcal{F}$ be an arbitrary two-party ideal functionality. Let $\mathcal{G}$ be a setup functionality, such as a CRS. Let Π be an $(\mathcal{F}_{\mathrm{ach}}, \mathcal{F}_{\mathrm{sch}})$-hybrid protocol that multi-realizes $\mathcal{F}$ with joint access to $\mathcal{G}$ (where $\mathcal{F}_{\mathrm{ach}}$ is defined in §2.5 and $\mathcal{F}_{\mathrm{sch}}$ is defined in §2.6).

Our goal is to use Π to design an $\mathsf{s}\mathcal{F}_{\mathrm{msc}}$-hybrid protocol $\mathsf{s}\Pi$ that multi-realizes $\mathsf{s}\mathcal{F}$ with joint access to $\mathcal{G}$. The point is, $\mathsf{s}\Pi$ does not require secure channels. Moreover, instantiating $\mathsf{s}\mathcal{F}_{\mathrm{msc}}$ with Π_{smsc}, we obtain the a protocol $\mathsf{s}\Pi[\mathsf{s}\mathcal{F}_{\mathrm{msc}}/\Pi_{\mathrm{smsc}}]$ that multi-realizes $\mathsf{s}\mathcal{F}$ with joint access to $\mathcal{G}$.

At a high level, $\mathsf{s}\Pi$ works as follows:

1. Wait for an input message `init`; then send the message `init` to $\mathsf{s}\mathcal{F}_{\mathrm{msc}}$.
2. Wait for a message (`authorze`, *chid*) from $\mathsf{s}\mathcal{F}_{\mathrm{msc}}$; then do the following:
   (a) initialize a "virtual" instance of Π, assigning it a PID and SID that are the same as that of this protocol instance, except that the SID pathname is prefixed *chid*;
   (b) generate the output message (`authorize`, *chid*).
3. Proceed as follows:
   (a) process input requests by passing them to the virtual instance of Π;
   (b) pass along outputs of the virtual instance of Π as outputs of this protocol instance;

(c) use $\mathsf{s}\mathcal{F}_{\text{msc}}$ to implement the secure channels used by the virtual instance of Π, translating the SIDs associated with the various secure channel instances to corresponding subsession IDs.

Theorem 3 *If Π is an $(\mathcal{F}_{\text{ach}}, \mathcal{F}_{\text{sch}})$-hybrid protocol that multi-realizes $\mathcal{F}$ with joint access to $\mathcal{G}$, then $\mathsf{s}\Pi$ is an $\mathsf{s}\mathcal{F}_{\text{msc}}$-hybrid protocol that multi-realizes $\mathsf{s}\mathcal{F}$ with joint access to $\mathcal{G}$.*

NOTES:

1. This is essentially the same as the main technical result (Lemma 4.1) of [BCL$^+$05], but with the following differences:

   (a) We assume that Π works with secure channels, while [BCL$^+$05] assumes authenticated channels. Since secure channels are easily built from authenticated channels (at least assuming erasures), this makes little difference from a theoretical perspective; however, from a practical point of view, it seems better to start with secure channels.

   (b) The name mangling is somewhat different; although this is mainly just a syntactic difference, with our specialized naming conventions, our formulation more readily allows Π to make use of subroutines, whereas with the formulation in [BCL$^+$05], it is not clear how this is to be done. In fact, since authenticated channels are actually implemented as subroutines, it is not clear what naming conventions are to be used for the SIDs of these subroutines in [BCL$^+$05].

   (c) The theorem in [BCL$^+$05] essentially says that if Π 2-realizes $\mathcal{F}$ with joint access to $\mathcal{G}$ (i.e., multi-realizes with a limit of just 2 instances), then $\mathsf{s}\Pi$ is an $(\mathsf{s}\mathcal{F}_{\text{msc}}, \mathcal{G})$-hybrid protocol that realizes $\mathsf{s}\mathcal{F}$. This is weaker (and much less useful) than the statement of our theorem, but the proof goes through without any difficulties.

2. The proof idea is fairly natural, even though the details are a bit involved:

   (a) Suppose we start with an adversary A that attacks $\mathsf{s}\Pi$.

   (b) From A, we can naturally derive an adversary $\tilde{A}$ that attacks Π.
   This attack essentially takes place in the "virtual world". In this virtual world, there may well be two parties with the same PID, one of whom is honest and one of whom is corrupt; however, these two parties will have different SIDs. Because of this, the notion of PID-wise corruption is not useful in this setting.

   (c) From the security property of Π, we obtain an equivalent adversary $\tilde{A}^*$ whose attack on $\mathcal{F}$ in our virtual world is indistinguishable from the attack of $\tilde{A}$ on Π.

   (d) From $\tilde{A}^*$, we naturally derive an adversary A^* whose attack on $\mathsf{s}\mathcal{F}$ is indistinguishable from the attack of A on $\mathsf{s}\Pi$.

3. Our restrictions in §2.3 were in part motivated by the issues raised in points 1b and 2b above.

5 UC zero knowledge

Before getting into strong CAID/CAKE protocols, we need to discuss an essential building block: practical protocols for UC ZK (zero knowledge).

5.1 The basic ideal functionality

Let R be a binary relation, consisting of pairs (x, w): for such a pair, x is called the "statement" and w is called the "witness".

The SID for a ZK protocol is of the form $\mathit{parent}/\mathit{ext} : P_{\mathrm{pid}} : Q_{\mathrm{pid}} :$, where P is the prover and Q the verifier. For an adversary A, an instance of $\mathcal{F}_{\mathrm{zk}}$ with such an SID runs as follows:

1. Wait for both:
   (a) an input message $(\mathtt{send}, x, w)$ from P such that $(x, w) \in R$; then send $(\mathtt{send}, x)$ to A;
   (b) an input message $\mathtt{ready}$ from Q; then send the message $\mathtt{ready}$ to A.
2. Wait for the message $\mathtt{deliver}$ from A; then send the output message $(\mathtt{deliver}, x)$ to Q.

Corruption rule: If P is corrupted between Steps 1a and 2, then A is given w, and is allowed to change the value of (x, w) to any value $(x', w') \in R$ (at any time before Step 2).

Note the similarity with the authenticated channels functionality $\mathcal{F}_{\mathrm{ach}}$ in §2.5. All that is different is that the ideal functionality guarantees that x comes with a corresponding witness.

5.2 An enhanced functionality

For our purposes, we need a slightly stronger form of security, which we call "enhanced" ZK. The basic idea is that we want to delay revealing the statement x to the verifier until the last possible moment (this helps prevent the simulator from being "over committed"). In addition, since we eventually use secure channels, we won't even reveal x to the adversary if neither prover nor verifier are corrupted.

Again, let R be a binary relation, consisting of pairs (x, w). Also, let $\ell : \{0, 1\}^* \to \{0, 1\}^*$ be an "information leakage" function. As for ZK, the SID for an enhanced ZK protocol is of the form $\mathit{parent}/\mathit{ext} : P_{\mathrm{pid}} : Q_{\mathrm{pid}} :$, where P is the prover and Q the verifier.

For an adversary A, an instance of $\mathcal{F}_{\mathrm{ezk}}$ with SID $\mathit{sid} := \mathit{parent}/\mathit{ext} : P_{\mathrm{pid}} : Q_{\mathrm{pid}} :$ runs as follows:

1. Wait for both:
   (a) an input message $(\mathtt{send}, x, w)$ from P such that $(x, w) \in R$; then send the message $(\mathtt{send}, \ell(x))$ to A;
   (b) an input message $\mathtt{ready}$ from Q; then send $\mathtt{ready}$ to A.
2. Wait for the message $\mathtt{lock}$ from A.
3. Wait for both:
   (a) a message $\mathtt{done}$ from A; then send the output message $\mathtt{done}$ to P.
   (b) a message $\mathtt{deliver}$ from A; then send the output message $(\mathtt{deliver}, x)$ to Q.

Corruption rule: If P is corrupted between Steps 1a and 2, then A is given (x, w) and is allowed to change the value of (x, w) to any value $(x', w') \in R$ (at any time before Step 2).

Note the similarity with our secure channels functionality. Here, the functionality is parameterized by the information leakage function ℓ, which is used to model the fact that some information about x may be leaked to an eavesdropping adversary. Typically, this information will be some rough information about the "size and shape" of x that ultimately determines the lengths of the ciphertexts that must be sent in an implementation.

As we will see, realizing $\mathcal{F}_{\mathrm{ezk}}$ is not really any harder than realizing $\mathcal{F}_{\mathrm{zk}}$.

5.3 Parameterized relations

In the above discussion, the relation R was considered to be a fixed relation. However, for many applications, it is convenient to let R be parameterized by a some system parameter (see §2.4). To realize the ZK (or extended ZK) functionality, it may be necessary to assume that the system parameter was generated in a certain way.

For example, a ZK protocol might require that the system parameter contains an RSA modulus N that is the product of two primes. To realize the ZK ideal functionality, it might not even be necessary that the factorization of N remain hidden. In such a case, the system parameter might be profitably viewed as a public-coin system parameter. This means that the environment may know the factorization of N, which may be useful to model situations where the factorization of N is used, say, to sign messages in higher-level protocols that use a ZK protocol as a subprotocol.

5.4 Relaxing the requirements: allowing a gap between soundness and zero knowledge

For some applications (such as IDEMIX "show proofs"), significant efficiency improvements are possible if one relaxes the requirements a bit. One might insist that witnesses supplied by honest provers are of a certain form, while allowing a dishonest prover to supply a witness of a more general form.

For example, perhaps an honest prover is required to supply an integer witness that lies in some interval — technically, this requirement may be necessary to ensure simulatability for an honest prover, i.e., zero knowledge. However, perhaps we wish to allow a dishonest prover to supply an integer witness in a slightly larger interval — technically, this is due to a limitation of the "knowledge extractor" that could perhaps be overcome, but at a significant cost to the efficiency of the protocol.

Formally, this relaxation is achieved by modifying the ZK ideal functionality — defining one relation R for honest provers, and another relation $R' \supseteq R$ for dishonest provers.

5.5 Practical UC ZK

Practical ZK protocols exist for the types of relations that we will be needed in our strong CAID/CAKE protocols — indeed, our protocols were designed with such protocols specifically in mind. In a companion paper, we give a detailed account of the current state of the art for such protocols. Here, we content ourselves with a brief summary.

We will be proving statements of the form

$$\text{Ӿ}\, w_1 \in \mathcal{D}_1, \ldots, w_n \in \mathcal{D}_n : \phi(w_1, \ldots, w_n). \tag{1}$$

Here, we use the symbol "Ӿ" instead of "∃" to indicate that we are proving "knowledge" of a witness, rather than just its existence. Here, the $\mathcal{D}_i$'s are domains which are finite sets of integers and ϕ is a predicate — we will presently place restrictions on the form of the domains and the predicate. A witness for a statement of the form (1) is a tuple $(w_1, \ldots, w_n)$ of integers such that $w_i \in \mathcal{D}_i^*$ for $i = 1 \,..\, n$ and $\phi(w_1, \ldots, w_n)$ — here, the each $\mathcal{D}_i^*$ is a finite set of integers that contains the domain $\mathcal{D}_i$ — we discuss this in greater detail below.

More on the predicate. The predicate $\phi(w_1, \ldots, w_n)$ is given by a formula that is built up from "atoms" using arbitrary combinations of ANDs and ORs. Each atom is of the form

$$\prod_{j=1}^{k} g_j^{F_j} = 1,$$

where the g_j's are elements of an abelian group, and the F_j's are integer polynomials in the variables $w_1, \ldots, w_n$.

More on groups. Different groups may be used in different atoms. The descriptions of the groups appearing in the predicate ϕ will in general be given as system parameters (see 2.4). One requirement is that the orders of groups contain no small prime factors (where "small" means less than an implementation-defined parameter, which is typically a 160–256 bit number); this requirement must be ensured by the trusted party that sets up the system parameters, if it cannot be verified directly; it is possible to relax this requirement, but we shall not discuss this possibility here. Another requirement is that encodings of group elements must be efficiently recognizable (in addition to having efficient algorithms for performing group operations, of course). It is not required that the discrete logarithm problem in the group is hard; however, if this assumption is allowed, it can be exploited for efficiency gains.

For our applications, we will mainly be working with two types of groups: groups of known prime order q, and groups of unknown composite order, which arise in anonymous credential systems such as IDEMIX. In the original presentation of the IDEMIX system, one works with the group $\mathbb{Z}_N^*$, or the subgroup of elements with Jacobi symbol 1, where N is the product of two strong primes (i.e., each prime is twice a prime plus 1). Unfortunately, the order of these groups is divisible by 2, violating our requirement stated above that group orders have no small prime divisors. Working with the group $(\mathbb{Z}_N^*)^2$ is not an option either, as this group is not efficiently recognizable. An elegant way to fix this problem is to use the group of signed quadratic residues, $(\mathbb{Z}_N^*/\{\pm 1\})^2$, which has no small prime divisors but is efficiently recognizable — this is yet another fruitful application of signed quadratic residues (see [HK09]). All the IDEMIX protocols are trivially adjusted to work with this group. Also, note that since a credential-issuing authority will be using the factorization of N to issue credentials, we formally view N as a *public-coins* system parameter. This ensures that our protocols retain their security properties in an environment in which credentials are being issued, without having to separately analyze the interaction between the credential-issuing protocol and the ZK protocol.

More on domains. In general, the domain of a variable is a set of integers bounded in absolute value by a given integer m:

$$\mathcal{B}(m) := \{w \in \mathbb{Z} : -m \leq w < m\}.$$

If a variable w_i is specified to lie in the domain $\mathcal{D}_i = \mathcal{B}(m)$, then an honest prover must supply $w \in \mathcal{D}_i$, while a dishonest prover is allowed to supply a witness in the domain $\mathcal{D}_i^* = \mathcal{B}(mt)$, where t is an implementation-defined parameter. This "gap" between honest provers and dishonest provers was discussed in §5.4. In typical implementations, t may be a 200–400 bit integer. Of course, if w_i only appears as an exponent of group elements whose orders are known to divide m, then we may assume that w_i is reduced modulo m, and thus take $t = 1$; in this case, we may simply write $\mathcal{D}_i$ and $\mathcal{D}_i^*$ as $\mathbb{Z}_m$.

Implementations. It is known how to construct efficient UC zero-knowledge protocols for these types of statements, under reasonable assumptions, by

(i) combining known techniques to obtain efficient Σ-protocols,

(ii) converting these Σ-protocols to corresponding Ω-protocols by essentially verifiably encrypting the witness,

(iii) and converting these Ω-protocols into efficient UC ZK protocols by using an appropriate type of trapdoor commitment scheme.

The techniques for (i) are essentially the (straightforward) generalization of Schnorr proofs [Sch91], the OR-proof technique [CDS94], and techniques for proving multiplicative relations among committed values [CS97, FO99]. The techniques for (ii) and (iii) are found in [MY04] — this yields practical $\mathcal{F}_{\text{ach}}$-hybrid protocols that multi-realize $\mathcal{F}_{\text{zk}}$ with joint access to a CRS; using the "committed proofs" idea from Jarecki and Lysyanskaya [JL00], we obtain practical $\mathcal{F}_{\text{sch}}$-hybrid protocols that multi-realize $\mathcal{F}_{\text{ezk}}$ with joint access to a CRS.

The computational complexity of these proof systems can be easily related to the arithmetic circuit complexity of the polynomials that appear in the description of ϕ: the number of exponentiations is proportional to the sum of the circuit complexities; a more precise running time estimate depends on the types of groups and domains.

Extensions. We have just defined a simple yet expressive language for practical UC ZK; we now extend the language in a number of ways. In principle, these extensions do not increase the expressibility of the language. However, these extensions are make it easier to directly and naturally express more complex relations, and more importantly, they make it easier for an "optimizing compiler" to generate much more efficient protocols. The extended language allows atoms that are predicates of the following type:

- $F = 0$;
- $F \geq 0$;
- $F \equiv 0 \pmod{m}$;
- $\gcd(F, m) = 1$;

here, F is an integer polynomial in the variables $w_1, \ldots, w_n$, and m is a positive integer. Note that the predicate $F = 0$ can be expressed in the base language by using the additive group $\mathbb{Z}$; the other three types of predicates are easily expressed using this type of predicate. In the companion paper, we discuss conditions under which the language may be extended more generally. We also discuss another extension to the language, which allows some variables to be quantified using $\exists$ rather than Ӿ. The idea is that witnesses quantified under $\exists$ are asserted just to exist, rather than to be explicitly "known" by the prover. Making sense of this formally requires some effort; however, the effort pays off in that the resulting ZK protocols may be substantially more efficient (as witnesses quantified under $\exists$ do not have to encrypted using Paillier encryption, which is quite expensive), and even those these protocols are not really UCZK, they may still be used in a modular way as building blocks in larger protocols.

6 Strong CAID/CAKE protocols

6.1 A protocol for vectored unions of product relations

We present here a protocol Π_0 for $\mathcal{F}^*_{\text{caid}}$ that works for a vectored union of product relations (see §3.4).

We assume the relation is described by values $x_1, \ldots, x_k$ and $y_1, \ldots, y_k$. Party P has inputs $s_1 \in S^*_1, \ldots, s_k \in S^*_k$, and Q has inputs $t_1 \in T^*_1, \ldots, t_k \in T^*_k$. They are trying to determine if

$$\bigvee_{i=1}^{k} \Big[(x_i, s_i) \in E_i \wedge (y_i, t_i) \in F_i \Big],$$

for fixed relations $E_1, \ldots, E_k$ and $F_1, \ldots, F_k$.

We assume that as system parameters, we have a group $\mathbb{G}$ of prime order q, and random generator g. We will need to assume that the computational Diffie-Hellman (CDH) assumption holds in this group. This protocol also requires some extra machinery, described below.

1a. P computes $h_{\mathrm{L}} \leftarrow_{\mathrm{R}} \mathbb{G}$ and sends h_{L} to Q over a secure channel.

1b. Q computes $h_{\mathrm{R}} \leftarrow_{\mathrm{R}} \mathbb{G}$ and sends h_{R} to P over a secure channel.

2a. P waits for h_{R}, and then computes:

```
for $i = 1 \mathinner{..} k$:
    $\alpha_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $\alpha'_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$
    if $(x_i, s_i) \in E_i$
        then $e_i \leftarrow g^{\alpha_i}$
        else $e_i \leftarrow h_{\mathrm{R}}/g^{\alpha'_i}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, P proves to Q:

$$\text{ꓘ}\,\{s_i \in S_i^*, \alpha'_i \in \mathbb{Z}_q\}_{i=1}^{k} : \left[\bigwedge_{i=1}^{k}\Big((x_i, s_i) \in E_i \vee g^{\alpha'_i} = h_{\mathrm{R}}/e_i\Big)\right];$$

Note that $e_1, \ldots, e_k$ are delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after P erases $\alpha'_1, \ldots, \alpha'_k$.

2b. Q waits for h_L, and then computes:

```
for $i = 1 \mathinner{..} k$:
    $\beta_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $\beta'_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$
    if $(y_i, t_i) \in E_i$
        then $f_i \leftarrow g^{\beta_i}$
        else $f_i \leftarrow h_{\mathrm{L}}/g^{\beta'_i}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, Q proves to P:

$$\text{ꓘ}\,\{t_i \in T_i^*, \beta'_i \in \mathbb{Z}_q\}_{i=1}^{k} : \left[\bigwedge_{i=1}^{k}\Big((y_i, t_i) \in E_i \vee g^{\beta'_i} = h_{\mathrm{L}}/f_i\Big)\right];$$

Note that $f_1, \ldots, f_k$ are delivered to P via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after Q erases $\beta'_1, \ldots, \beta'_k$.

3a. P computes:

```
for $i = 1 \mathinner{..} k$:
    if $(x_i, s_i) \in E_i$
        then $u_i \leftarrow f_i^{\alpha_i}$
        else $u_i \leftarrow_{\mathrm{R}} \mathbb{G}$
```

3b. Q computes:

```
for $i = 1 \mathinner{..} k$:
    if $(y_i, t_i) \in E_i$
        then $v_i \leftarrow e_i^{\beta_i}$
        else $v_i \leftarrow_{\mathrm{R}} \mathbb{G}$
```

4. P and Q run a strong CAID subprotocol to evaluate the predicate

$$\bigvee_{i=1}^{k} (u_i = v_i),$$

and output the result of this computation after erasing all local data.

NOTES:

1. The values h_{L} and h_{R} could be replaced by a system parameter $h \in \mathbb{G}$.

2. We have reduced our original strong CAID problem to a simpler strong CAID problem in Step 4. We discuss implementations of Step 4 below.

3. The intuition for the main idea of the protocol runs as follows. Suppose, for example, that P is honest and Q is corrupt. In Step 2b, Q intuitively proves for each $i = 1 \mathinner{..} k$, either that it knows t_i such that $(y_i, t_i) \in E_i$ or that it *does not know* β_i; in the latter case, Q will not be able to predict the value $g^{\alpha_i \beta_i}$ when it comes to Step 4.

4. Assuming the E_i's and F_i's are relations based on an anonymous credential system like IDEMIX, then all of the ZK protocols have relatively efficient implementations (see §5.5).

6.2 Security analysis

Our goal now is to show that protocol Π_0 realizes $\mathcal{F}^*_{\mathrm{caid}}$. Note that Π_0 is a hybrid protocol that uses the following ideal functionalities as subroutines: secure channels (i.e., $\mathcal{F}_{\mathrm{sch}}$), enhanced zero knowledge (i.e., $\mathcal{F}_{\mathrm{ezk}}$) for relations of the form appearing in Steps 2a and 2b of the protocol, and $\mathcal{F}^*_{\mathrm{caid}}$ for relations of the form appearing in Step 4 of the protocol.

We need to make use of the CDH assumption. It will be helpful to introduce some notation. For group elements $u = g^\alpha$ and $v = g^\beta$ in $\mathbb{G}$, where $\alpha, \beta \in \mathbb{Z}_q$, let us define $[u, v] := g^{\alpha\beta}$. The CDH assumption says that for random $u, v \in \mathbb{G}$, it is hard to compute $[u, v]$.

Theorem 4 *Under the CDH assumption for $\mathbb{G}$, protocol Π_0 realizes $\mathcal{F}^*_{\mathrm{caid}}$.*

Proof. As usual, our goal is as follows: given an adversary A attacking Π_0, construct a corresponding simulator A^* that attacks $\mathcal{F}^*_{\mathrm{caid}}$, so that no environment can distinguish an attack of A from the an attack of A^*. Moreover, the design of A^* will follow the usual principle of running an instance of A internally, and providing to A a simulated execution environment, including all of the ideal functionalities used as subroutines in Π_0.

We are assuming here that A attacks just a single instance of Π_0. Because protocol Π_0 uses secure channels and enhanced zero knowledge, in the case where both P and Q are honest, the simulation is trivial — A^* just sends and receives various notifications to A, and these notifications contain no real content; in addition, when A send a `lock`, `deliver-left`, or `deliver-right` messages for the $\mathcal{F}^*_{\mathrm{caid}}$ functionality in Step 4, A^* sends a corresponding message to its own $\mathcal{F}^*_{\mathrm{caid}}$ functionality.

The simulation only becomes challenging when one of P and Q becomes corrupt. We only consider the case Q becomes corrupted first (the case where P is corrupted first is symmetric). Without loss of generality, we may assume that as soon as Q is corrupted, all of its subroutines are immediately corrupted as well. In fact, it suffices to consider the case where Q (as well as all of its subroutines) starts out corrupt: because of the use of secure channels and enhanced zero knowledge, after obtaining Q's input $t_1, \ldots, t_k$, the simulator can simply start Q's execution of Π_0 at the beginning, and run it forward to the point where Q was actually corrupted. Of course, our simulator has to deal with the possibility that P may be corrupted at any time, and be able to produce a convincing view of P's internal state at that time (and at which point in time, the simulators task is complete).

Step 1. A^* follows the protocol, generating $h_{\mathrm{L}} \in \mathbb{G}$ at random, and sending this to Q (i.e., the real-world Q that is under the control of A).

Step 2. When Q performs his ZK proof, he gives to A^* witnesses t_i, β'_i for $i = 1 \mathinner{\ldotp\ldotp} k$, along with the values $f_1, \ldots, f_k$ (which are sent directly to P).

Now consider P's ZK proof. If P is corrupted before the "lock" step of $\mathcal{F}_{\mathrm{ezk}}$, then A^* obtains $s_1, \ldots, s_k$ from $\mathcal{F}^*_{\mathrm{caid}}$, and computes the internal state of P as in the protocol.

Otherwise, if P is not corrupted before the "lock" step of $\mathcal{F}_{\mathrm{ezk}}$, then A^* computes $\alpha_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $e_i \leftarrow g^{\alpha_i}$ for $i = 1 \mathinner{\ldotp\ldotp} k$, and delivers to Q the x_i's and e_i's which describe the statement being proved.

If P is ever corrupted at any later point in time, A^* obtains $s_1, \ldots, s_k$, and gives these values to A. In addition, for $i = 1 \mathinner{\ldotp\ldotp} k$ with $(x_i, s_i) \in E_i$, if $(x_i, s_i) \notin E$, then A^* replaces α_i by a random element of $\mathbb{Z}_q$, before sending $(\alpha_1, \ldots, \alpha_k)$ to A. Notice that A^* does not need to send any of the α'_i values to A: we are exploiting here in an essential way the fact that P erases these values, and the properties of the $\mathcal{F}_{\mathrm{ezk}}$ functionality, which does not commit us to the values $e_1, \ldots, e_k$ until after the values $\alpha'_1, \ldots, \alpha'_k$ are erased.

Step 3. This is a local computation, so there is nothing to simulate. However, if P is corrupted after this step, then A^* obtains $s_1, \ldots, s_k$, and in addition to these values, gives $\alpha_1, \ldots, \alpha_k$ and $u_1, \ldots, u_k$ to A, where the α_i's are computed as above, and the u_i's are computed as follows: if $(x_i, s_i) \in E$, then $u_i \leftarrow f_i^{\alpha_i}$, else $u_i \leftarrow_{\mathrm{R}} \mathbb{G}$.

Step 4. After A "locks" in the inputs to the $\mathcal{F}^*_{\mathrm{caid}}$ subprotocol, the simulator A^* obtains Q's inputs $v_1, \ldots, v_k$ to the subprotocol. Because Q is malicious, these inputs may not be correctly computed, and so A^* will adjust the values $t_1, \ldots, t_k$ obtained in Step 2 accordingly. Specifically, for $i = 1 \mathinner{\ldotp\ldotp} k$, if $(y_i, t_i) \in F_i$ but $v_i \neq f_i^{\alpha_i}$, then A^* replaces t_i with $\perp$. Recall that by convention, $(y_i, \perp) \notin F_i$. Now A^* submits the adjusted inputs $t_1, \ldots, t_k$ to the high level $\mathcal{F}^*_{\mathrm{caid}}$ functionality, and delivers the result to Q.

That completes the description of A^*. Clearly, A^* perfectly simulates A's attack, except for one possibility — namely, the output of the high-level $\mathcal{F}^*_{\mathrm{caid}}$ functionality may not agree with the output of the low-level $\mathcal{F}^*_{\mathrm{caid}}$ functionality. We argue that this happens with negligible probability, under the CDH assumption.

Let $e_1, \ldots, e_k$ and $\alpha_1, \ldots, \alpha_k$ be computed as described in Step 2 above. Let $v_1, \ldots, v_k$ and $t_1, \ldots, t_k$ be computed as described in Step 4 above. Let $t'_1, \ldots, t'_k$ denote the original, unadjusted t_i values, as obtained in Step 2. Let $u_1, \ldots, u_k$ be computed as follows: if $(x_i, s_i) \in E$, then $u_i \leftarrow f_i^{\alpha_i}$, else $u_i \leftarrow_{\mathrm{R}} \mathbb{G}$. Our goal is to show that with overwhelming probability, for $i = 1 \mathinner{\ldotp\ldotp} k$, $u_i = v_i$ if and only if $(x_i, s_i) \in E_i$ and $(y_i, t_i) \in F_i$.

Fix an index $i = 1 \mathinner{\ldotp\ldotp} k$.

If $(x_i, s_i) \in E_i$ and $(y_i, t_i) \in F_i$, then by definition, $u_i = f_i^{\alpha_i} = v_i$.

Now suppose $(x_i, s_i) \notin E_i$ or $(y_i, t_i) \notin F_i$. We want to show that $u_i \neq v_i$ with overwhelming probability. There are two cases to consider.

Case 1. $(x_i, s_i) \notin E_i$. In this case, u_i is random, and so $u_i \neq v_i$ with overwhelming probability.

Case 2. $(x_i, s_i) \in E_i$ and $(y_i, t_i) \notin F_i$. In this case, $u_i = f_i^{\alpha_i}$. There are two subcases.

Case 2a. $(y_i, t'_i) \in F_i$. In this case, $u_i \neq v_i$ by definition.

Case 2b. $(y_i, t'_i) \notin F_i$. This is the interesting case. In Step 2, as a part of Q's ZK proof, we obtained the value β'_i, and by definition, $g^{\beta'_i} = h_{\mathrm{L}}/f_i$. Also by definition, $u_i = [e_i, f_i]$. We assume that A can compute v_i such that $u_i = v_i$ with non-negligible probability, and obtain a contradiction. Note that

$$[e_i, h_{\mathrm{L}}] = [e_i, f_i g^{\beta'_i}] = [e_i, f_i][e_i, g^{\beta'_i}] = [e_i, f_i] e_i^{\beta'_i}.$$

The identity

$$[e_i, h_{\mathrm{L}}] = [e_i, f_i] e_i^{\beta'_i} \tag{2}$$

is the key identity. If A is able to compute $[e_i, f_i]$, then we can use A to compute $[e_i, h_{\mathrm{L}}]$. Moreover, in order to compute $[e_i, h_{\mathrm{L}}]$ in this way, we do not need the value α_i. Thus, we can use A to solve a random instance of the CDH problem with non-negligible probability, which contradicts the CDH assumption. □

6.3 Implementing Step 4

In the case where $k = 1$, one can use the equality test protocol in §7. As an alternative to protocol Π_0, in the case where $k = 1$ one can use the protocol in §8.

In the general case where $k \geq 1$, we suggest the following method. Assume we have a UC protocol for evaluating an arithmetic circuit mod N, where N is a system parameter that is the product of two large primes. Then to evaluate the boolean expression

$$\bigvee_{i=1}^{k} (u_i = v_i), \tag{3}$$

P chooses $a_0 \in \mathbb{Z}_N$ at random, and for $i = 1\,..\,k$, encodes u_i as an element a_i of $\mathbb{Z}_N$; similarly, Q chooses $b_0 \in \mathbb{Z}_N$ at random, and for $i = 1\,..\,k$, encodes v_i as an element b_i of $\mathbb{Z}_N$. Then P and Q jointly evaluate in the expression (over $\mathbb{Z}_N$)

$$\prod_{i=0}^{k} (a_i - b_i). \tag{4}$$

If (3) is true, then (4) is zero; if (3) is false, then (4) is a random element of $\mathbb{Z}_N$.

Thus, we reduce the original strong CAID problem to a strong CAID problem for a simpler predicate, namely, expressions of the type (3), and the latter is easily reduced to a simple circuit evaluation problem modulo N, namely, expressions of the type (4). There are quite practical protocols for circuit evaluation, which we discuss in detail in a companion paper. The basic idea is to use known techniques for circuit evaluation based on homomorphic encryption, making use of a semantically secure variant of Camenisch and Shoup's encryption scheme [CS03], which has the advantage that generating public keys is very inexpensive (making security with adaptive corruptions and erasures more practical) and proofs about plaintexts fit very nicely into the framework for ZK proofs discussed in §5.5. These protocols require $O(k)$ exponentiations, and $O(\log k)$ (the circuit depth) rounds of communication, and $O(k)$ total communication complexity. These same bounds hold for the overall strong CAID protocol as well.

6.4 Variations

Using a hash. Note that in implementing Step 4, it may be more convenient if protocol Π_0 computed the u_i's and v_i's as the *hash* of group elements. This will work, provided we assume the DDH, and provided the hash function is sufficiently entropy preserving.

Realizing $\mathcal{F}_{\mathrm{pcaid}}$. To realize $\mathcal{F}^*_{\mathrm{pcaid}}$, we simply add the following step before Step 1:

0.1a P waits for its input $\langle R \rangle$, and then P sends $\langle R \rangle$ to Q over a secure channel.

0.1b Q waits for the message $\langle R \rangle$ from P, and then outputs $\langle R \rangle$ After receiving the message $\langle R \rangle$ from P, Q outputs $\langle R \rangle$.

0.2 P (resp., Q) waits for the input s (resp., t).

This is a generic transformation that converts any $\mathcal{F}^*_{\text{caid}}$ protocol to an $\mathcal{F}^*_{\text{pcaid}}$ protocol.

Adding key exchange. Adding key exchange is simple, especially since we are already assuming secure channels. We simply add the following step before Step 1:

0.3 P sends K to Q over a secure channel.

In addition, whenever either party would output 1, it instead outputs $(1, K)$.

This is a generic transformation that converts any $\mathcal{F}^*_{\text{caid}}$ (resp., $\mathcal{F}^*_{\text{pcaid}}$) protocol into an $\mathcal{F}^*_{\text{cake}}$ (resp., $\mathcal{F}^*_{\text{pcake}}$) protocol.

Adding partial equality tests. Suppose that we want to add support for partial equality tests (see §3.4). The relation we now want to work with is

$$\Big\{ ((s_1, \ldots, s_k), (t_1, \ldots, t_k)) : \bigvee_{i=1}^{k} \Big[(x_i, s_i) \in E_i \wedge (y_i, t_i) \in F_i \wedge (\sigma_i(s_i) = \tau_i(t_i)) \Big] \Big\},$$

for appropriate functions $\sigma_1, \ldots, \sigma_k$ and $\tau_1, \ldots, \tau_k$. To do this, we need to make use of a variation of the "commit and prove" functionality introduced in [CLOS02]. More specifically, in Steps 2a and 2b, P should make commitments to $u'_i = \sigma_i(s_i)$ for $i = 1 \mathinner{..} k$ and Q should make commitments to $v'_i := \tau_i(t_i)$ for $i = 1 \mathinner{..} k$, in such a way that both parties are committed to use these same values in Step 4. Then, in Step 4, the two parties run a strong CAID protocol to evaluate the predicate

$$\bigvee_{i=1}^{k} \big[(u_i = v_i) \wedge (u'_i = v'_i) \big]. \tag{5}$$

Evaluation of this predicate can also be reduced to a circuit evaluation problem over $\mathbb{Z}_N$. Assuming u_i, v_i, u'_i, v'_i are encoded as elements of $\mathbb{Z}_N$ as a_i, b_i, a'_i, b'_i, P chooses random $\mu_i \in \mathbb{Z}_N$ for $i = 0 \mathinner{..} k$; similarly, Q chooses random $\nu_i \in \mathbb{Z}_N$ for $i = 0 \mathinner{..} k$. Then both parties jointly evaluate

$$(\mu_0 - \nu_0) \prod_{i=1}^{k} \big[(a_i - b_i)(\mu_i - \nu_i) + (a'_i - b'_i) \big].$$

Zero means (5) holds, and non-zero means it does not hold.

6.5 From strong CAID/CAKE to CAID/CAKE

We can instantiate protocol Π_0 to get a practical $\mathcal{F}_{\text{sch}}$-hybrid protocol Π'_0 that multi-realizes $\mathcal{F}^*_{\text{caid}}$ (or any of the variations discussed above) with joint access to a CRS — the crucial building block is $\mathcal{F}_{\text{ezk}}$, discussed in §5. Then using the split functionalities techniques in §4, we can turn Π'_0 into a protocol $\mathsf{s}\Pi'_0$ that multi-realizes $\mathsf{s}\mathcal{F}^*_{\text{caid}}$ with joint access to a CRS. The resulting protocol is a CAID/CAKE protocol that works without secure channels.

Typically, the purpose of running a CAKE protocol is to use the session key to implement a secure session. If, in fact, this is the goal, a more straightforward way of achieving it is as follows. Simply design a $\mathcal{F}_{\text{sch}}$-hybrid protocol that works as follows: first, it runs a strong CAID protocol, and if that succeeds, the parties continue to communicate, using the secure channels provided by the $\mathcal{F}_{\text{sch}}$ functionality. Now apply the split functionalities techniques in §4 to this protocol, obtaining a protocol that essentially provides a "credential authenticated secure channel".

7 A protocol for equality testing and a related problem

Here is a simple protocol, Π_{eq}, for equality testing. We assume that a group $\mathbb{G}$ of prime order q, along with a generator $g \in \mathbb{G}$, are given as system parameters. We will need to assume the DDH for $\mathbb{G}$. We assume the inputs to the two parties are encoded as elements of $\mathbb{Z}_q$. Again, we use $\mathcal{F}_{\mathrm{ezk}}$ as a subprotocol.

The protocol runs as follows, where P has input $a \in \mathbb{Z}_q$, and Q has input $b \in \mathbb{Z}_q$:

1. P computes:

   $$h \leftarrow_{\mathrm{R}} \mathbb{G},\ x_1, x_2, r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$$
   $$c \leftarrow g^{x_1} h^{x_2},\ u_1 \leftarrow g^r,\ u_2 \leftarrow h^r,\ e \leftarrow g^a c^r$$

   and using $\mathcal{F}_{\mathrm{ezk}}$ proves to Q:

   $$\rtimes\, a \in \mathbb{Z}_q\, \exists\, r \in \mathbb{Z}_q : g^r = u_1 \wedge h^r = u_2 \wedge g^a c^r = e;$$

   note that h, c, u_1, u_2, e are delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing r.

2. Q computes:

   $$s \leftarrow_{\mathrm{R}} \mathbb{Z}_q^*,\ t \leftarrow_{\mathrm{R}} \mathbb{Z}_q$$
   $$\tilde{u}_1 \leftarrow u_1^s g^t,\ \tilde{u}_2 \leftarrow u_2^s h^t,\ \tilde{e} \leftarrow e^s g^{-bs} c^t$$

   and using $\mathcal{F}_{\mathrm{ezk}}$ proves to P:

   $$\rtimes\, b \in \mathbb{Z}_q\, \exists\, s, t \in \mathbb{Z}_q : u_1^s g^t = \tilde{u}_1 \wedge u_2^s h^t = \tilde{u}_2 \wedge e^s g^{-bs} c^t = \tilde{e} \wedge \gcd(s, q) = 1;$$

   note that $\tilde{u}_1$, $\tilde{u}_2$, $\tilde{e}$ are delivered to P via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing s, t.

3. P computes:

   $$z \leftarrow_{\mathrm{R}} \mathbb{Z}_q^*$$
   $$d \leftarrow \tilde{e}^z (\tilde{u}_1)^{-zx_1} (\tilde{u}_2)^{-zx_2}$$

   and using $\mathcal{F}_{\mathrm{ezk}}$ proves to Q:

   $$\exists\, x_1, x_2, z \in \mathbb{Z}_q : g^{x_1} h^{x_2} = c \wedge \tilde{e}^z (\tilde{u}_1)^{-zx_1} (\tilde{u}_2)^{-zx_2} = d \wedge \gcd(z, q) = 1;$$

   here, d is delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing x_1, x_2, z.

4. After erasing all local data, both parties output 1 if $d = 1$, and output 0 otherwise.

NOTES:

1. We are using $\exists$ as well as $\rtimes$ quantifiers here. This allows for certain optimizations, since values quantified under $\exists$ are never explicitly needed in the simulator in the security proof below. However, in terms of understanding the basic protocol and its security, the reader may simply treat $\exists$ the same as $\rtimes$.

2. In Step 1, (h, c) is the public key and (x_1, x_2) the private key for "Cramer-Shoup Ultra-Lite" — the semantically secure version of Cramer-Shoup encryption. (u_1, u_2, e) is an encryption of g^a. We will exploit the fact that this scheme is "receiver non-committing", as was demonstrated by Jarecki and Lysyanskaya [JL00]. This property will allow us to simulate adaptive corruptions.

3. In Step 2, assuming (u_1, u_2, e) encrypts g^a, then $(\tilde{u}_1, \tilde{u}_2, \tilde{e})$ is a random encryption of $g^{s(a-b)}$.
4. In Step 3, P is decrypting $(\tilde{u}_1, \tilde{u}_2, \tilde{e})$, and raising it to the power z, so that $d = g^{zs(a-b)}$
5. All of the ZK protocols have practical implementations, as discussed in §5.5.

Theorem 5 *Assuming the DDH for $\mathbb{G}$, protocol Π_{eq} realizes functionality $\mathcal{F}^*_{\text{caid}}$ for the equality relation.*

Proof. Similarly to what was argued in the proof of Theorem 4, it suffices to exhibit a simulator in the case where P starts out honest and Q starts out corrupt, and the case where P starts out corrupt and Q starts out honest (unlike in the Theorem 4, the two cases are not symmetric).

Case 1: P honest, Q corrupt. The simulator runs as follows.

Step 1. If P is corrupted before the "lock" step of $\mathcal{F}_{\text{ezk}}$, then the simulator obtains P's input a, and computes the internal state of P as in the protocol. Otherwise, the simulator computes

$$m_1 \leftarrow_{\text{R}} \mathbb{Z}_q,\ m_2 \leftarrow_{\text{R}} \mathbb{Z}_q,\ w \leftarrow_{\text{R}} \mathbb{Z}_q \setminus \{0\},\ r_1 \leftarrow_{\text{R}} \mathbb{Z}_q,\ r_2 \leftarrow_{\text{R}} \mathbb{Z}_q \setminus \{r_1\}$$

and then

$$h \leftarrow g^w,\ c \leftarrow g^{m_1},\ u_1 \leftarrow g^{r_1},\ u_2 \leftarrow g^{r_2},\ e \leftarrow g^{m_2}$$

and delivers h, c, u_1, u_2, e to Q as in $\mathcal{F}_{\text{ezk}}$.

If P is corrupted at any later time, the simulator must reveal the internal state of P, which consists of a, x_1, x_2. The value a is obtained from the $\mathcal{F}^*_{\text{caid}}$ ideal functionality. The values x_1, x_2 are computed as solutions to the following equation:

$$\begin{pmatrix} 1 & w \\ r_1 & r_2 w \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 - a \end{pmatrix}$$

Step 2. After receiving the values b, s, t, along with $\tilde{u}_1, \tilde{u}_2, \tilde{e}$ from Q as a part of $\mathcal{F}_{\text{ezk}}$, the simulator submits the value b to the $\mathcal{F}^*_{\text{caid}}$ ideal functionality as Q's input, and then obtains the output *res* of $\mathcal{F}^*_{\text{caid}}$.

Step 3. If P is corrupted before the "lock" step of $\mathcal{F}_{\text{ezk}}$, then the simulator obtains a, x_1, x_2 as in the corruption handling logic in Step 1, chooses $z \in \mathbb{Z}^*_q$ at random, and reveals the internal state of P, which consists at this point of (a, x_1, x_2, z). Otherwise, the simulator computes $d \in \mathbb{G}$ as follows: if $res = 1$, then $d \leftarrow 1$, else $d \leftarrow_{\text{R}} \mathbb{G} \setminus \{1\}$. Finally, the simulator simply waits for the protocol to finish, and then instructs $\mathcal{F}^*_{\text{caid}}$ to deliver the output *res* to P.

That completes the description of the simulator for Case 1. We leave it to the reader to verifier that the simulation is indistinguishable from a real execution of Π_{eq}, assuming the DDH: the DDH is used to argue that Step 1 of the simulation is indistinguishable from a real execution; given a "test tuple" (g, h, u_1, u_2), where either $\log_g u_1 = \log_h u_1$ or u_1 and u_2 are random, we set up the system parameters with the given g; in Step 1, we take the input a from P, choose x_1, x_2 at random, and compute $c \leftarrow g^{x_1} h^{x_2}$, $e = g^a u_1^{x_1} u_2^{x_2}$; then we send h, c, u_1, u_2, e to Q, revealing x_1, x_2 if P is corrupted.

Case 2: P corrupt, Q honest. The simulator runs as follows.

Step 1. After receiving the values a, r along with h, c, u_1, u_2, e as a part of $\mathcal{F}_{\text{ezk}}$, the simulator submits the value a to the $\mathcal{F}^*_{\text{caid}}$ ideal functionality as P's input, and then obtains the output *res* of $\mathcal{F}^*_{\text{caid}}$.

Step 2. If Q is corrupted before the "lock" step of $\mathcal{F}_{\text{ezk}}$, then the simulator obtains Q's input and computes the internal state of Q as in the protocol. Otherwise, the simulator delivers to P

the values $\tilde{u}_1, \tilde{u}_2, \tilde{e}$ as part of $\mathcal{F}_{\mathrm{ezk}}$, which it computes as follows: if $res = 1$, then $(\tilde{u}_1, \tilde{u}_2, \tilde{e})$ is computed as a random encryption of 1, else $(\tilde{u}_1, \tilde{u}_2, \tilde{e})$ is computed as a random encryption of a random element of $\mathbb{G} \setminus \{1\}$.

Step 3. The simulator simply waits for the protocol to finish, and then instructs $\mathcal{F}^*_{\mathrm{caid}}$ to deliver the output *res* to Q.

That completes the description of the simulator in Case 2. We leave it to the reader to verify that the simulation is indistinguishable from a real execution of Π_{eq} (unconditionally). □

Applications. One application of protocol Π_{eq} is in the implementation of Step 4 of protocol Π_0 (see §6.3). However, in this situation, the protocol in §8 is more efficient.

Another application is to PAKE protocols. We can efficiently implement $\mathcal{F}_{\mathrm{ezk}}$ for the necessary relations using secure channels and a common reference string, and augment the protocol to share a random key over a secure channel. This gives us a fairly efficient strong CAKE protocol for the equality relation that uses secure channels. We then derive the split version of the protocol, using a simple Diffie-Hellman key exchange as in §4, which realizes the CAKE functionality (more precisely, it multi-realizes the CAKE functionality with joint access to a CRS). As observed in [BCL+05], a protocol that realizes this functionality in fact realizes the PAKE functionality (as defined in [CHK+05]). Our particular protocol is probably a bit less efficient than the one in [CHK+05]; however, our protocol has the advantage of being secure against adaptive corruptions (assuming erasures). A very different PAKE protocol, with a structure similar to that in [CHK+05], that is secure against adaptive corruptions was recently presented in [ACP09].

7.1 A variation

A variation on the above protocol gives a strong CAID protocol for the relation $DL := \{(a, g^a) : a \in \mathbb{Z}_q\}$. That is, it tests if $g^a = v$, where a is the input to P and v is the input to Q. The idea is to have Q "verifiably encrypt" v.

The protocol, which we call Π_{dl}, runs as follows:

1. P computes:

   $$h \leftarrow_{\mathrm{R}} \mathbb{G},\ x_1, x_2, r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$$
   $$c \leftarrow g^{x_1} h^{x_2},\ u_1 \leftarrow g^r,\ u_2 \leftarrow h^r,\ e \leftarrow g^a c^r$$

   and using $\mathcal{F}_{\mathrm{ezk}}$ proves to Q:

   $$\exists\!\!\!\!\!\!\;\;\rtimes a \in \mathbb{Z}_q\, \exists\, r \in \mathbb{Z}_q : g^r = u_1 \wedge h^r = u_2 \wedge g^a c^r = e;$$

   note that h, c, u_1, u_2, e are delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing r.

2. Q computes:

   $$s \leftarrow_{\mathrm{R}} \mathbb{Z}_q^*,\ t \leftarrow_{\mathrm{R}} \mathbb{Z}_q,\ y \leftarrow_{\mathrm{R}} \mathbb{Z}_q$$
   $$\tilde{v} \leftarrow g^y v,\ \tilde{u}_1 \leftarrow u_1^s g^t,\ \tilde{u}_2 \leftarrow u_2^s h^t,\ \tilde{e} \leftarrow e^s v^{-s} c^t$$

   and using $\mathcal{F}_{\mathrm{ezk}}$ proves to P:

   $$\rtimes y \in \mathbb{Z}_q\, \exists\, s, t \in \mathbb{Z}_q : u_1^s g^t = \tilde{u}_1 \wedge u_2^s h^t = \tilde{u}_2 \wedge e^s \tilde{v}^{-s} g^{ys} c^t = \tilde{e} \wedge \gcd(s, q) = 1;$$

   note that $\tilde{v}, \tilde{u}_1, \tilde{u}_2, \tilde{e}$ are delivered to P via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing y, s, t.

3. P computes:

$z \leftarrow_{\mathrm{R}} \mathbb{Z}_q^*$
$d \leftarrow \tilde{e}^z(\tilde{u}_1)^{-zx_1}(\tilde{u}_2)^{-zx_2}$

and using $\mathcal{F}_{\mathrm{ezk}}$ proves to Q:

$$\exists\, x_1, x_2, z \in \mathbb{Z}_q : g^{x_1}h^{x_2} = c \wedge \tilde{e}^z(\tilde{u}_1)^{-zx_1}(\tilde{u}_2)^{-zx_2} = d \wedge \gcd(z, q) = 1;$$

here, d is delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after erasing x_1, x_2, z.

4. Both parties output 1 if $d = 1$, and output 0 otherwise.

NOTES:

- Step 1 is exactly the same as before.
- In Step 2, Q is generating a random encryption of $(g^a/v)^s$. Moreover, by giving $\tilde{v}$ and y to $\mathcal{F}_{\mathrm{ezk}}$, Q is effectively giving v to $\mathcal{F}_{\mathrm{ezk}}$.
- Step 3 is the same as before, but now $d = (g^a/v)^{sz}$.

Theorem 6 *Assuming the DDH for $\mathbb{G}$, protocol Π_{dl} realizes functionality $\mathcal{F}^*_{\mathrm{caid}}$ for the relation DL.*

Proof. The proof is almost identical to the proof of Theorem 5. The main difference is in the way simulator works in Step 2: in Case 1, when the simulator receives y, s, t along with $\tilde{v}, \tilde{u}_1, \tilde{u}_2, \tilde{e}$ from Q as part of $\mathcal{F}_{\mathrm{ezk}}$, the simulator sets Q's input to $\mathcal{F}^*_{\mathrm{caid}}$ to the value $v := \tilde{v}/g^y$; in Case 2, the simulator generates $\tilde{v} \in \mathbb{G}$ at random. □

Applications. This protocol, when augmented with a key sharing step over a secure channel, and "split" as in §4, gives us a practical PAKE protocol that is secure against adaptive corruptions *and server compromise.* That is, the client stores the password a, while the server stores g^a. If the password file on the server is compromised, then it will not be easy to an attacker to login to the server as the client.

Unlike previous protocols, such as in [GMR06], our protocol does not rely on random oracles. To be fair, the definition of security in [GMR06] is so strong that it probably cannot be achieved without random oracles: the security definition in [GMR06] requires that in the event of a server compromise, an attacker must carry out an offline dictionary attack in order to guess the password. Also, note that the protocol in [GMR06] is proved secure only in the static corruption model.

In a complete PAKE protocol, one would likely set

$$a := H(\mathit{pw}, \mathit{clientID}, \mathit{serverID}),$$

where H is a cryptographic hash, *pw* is the actual password, and *clientID* and *serverID* are the names of the client and server, respectively. If H is entropy preserving, *pw* is a high-entropy password, and the discrete logarithm problem in $\mathbb{G}$ is hard, then it will be infeasible to login as the client, even if the server is compromised. Moreover, if H is modeled as a random oracle, and the discrete logarithm problem in $\mathbb{G}$ is hard, then even in the event of a server compromise, an attacker must still carry out an offline dictionary attack in order to login as the client.

Thus, our new protocol is the first fairly practical PAKE protocol (UC-secure or otherwise) that is secure against server compromise and does not rely on random oracles; as a bonus, it is also secure against adaptive corruptions.

8 A protocol for product relations

We present here a protocol Π_1 for $\mathcal{F}^*_{\mathrm{caid}}$ that works for a product relation $R = S \times T$. We assume party P executes the protocol with an input s and Q executes the protocol with input t. Protocol Π_1 is somewhat more efficient than combining protocol Π_0 in §6.1 with protocol Π_{eq} in §7.

We assume that the set S is determined by a value x and binary relation E, E so that $S = \{s \in S^* : (x, s) \in E\}$. Here, S^* is a fixed, efficiently recognizable set, and E is a fixed, efficiently computable relation. Similarly, T is determined by a value y and a binary relation F, so that $T = \{t \in T^* : (y, t) \in F\}$. Thus, S is the set of witnesses for x (according to the relation E) and T is the set of witnesses for y (according to the relation F). The relation R is described by the pair (x, y).

Let $\mathbb{G}$ be a group of large prime order q. We will assume that the DDH holds in $\mathbb{G}$. Let g be a random generator for $\mathbb{G}$. We assume that $\langle \mathbb{G}, q, g \rangle$ is a system parameter.

Our strong CAID protocol runs as follows. Here, both P and Q take as a common input the pair (x, y), via the SID; P takes input s and Q takes input t.

1a. P computes $h_{\mathrm{L}} \leftarrow_{\mathrm{R}} \mathbb{G}$ and sends h_{L} to Q over a secure channel.

1b. Q computes $h_{\mathrm{R}} \leftarrow_{\mathrm{R}} \mathbb{G}$ and sends h_{R} to P over a secure channel.

2a. P waits for h_{R}, and then computes:

```
for $i = 1..3$: $\alpha_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $\alpha'_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$
if $(x, s) \in E$
    then for $i = 1..3$: $e_i \leftarrow g^{\alpha_i}$
    else  for $i = 1..3$: $e_i \leftarrow h_{\mathrm{R}}/g^{\alpha'_i}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, P proves to Q

$$\text{⋊}\, \alpha'_1, \alpha'_2, \alpha_3 \in \mathbb{Z}_q,\ s \in S^* : \Big[g^{\alpha'_1} = h_{\mathrm{R}}/e_1 \wedge g^{\alpha'_2} = h_{\mathrm{R}}/e_2\Big] \vee \Big[g^{\alpha_3} = e_3 \wedge (x, s) \in E\Big];$$

Note that e_1, e_2, e_3 are delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after P erases $\alpha'_1, \alpha'_2, \alpha_3$.

2b. Q waits for h_{L}, and then computes:

```
for $i = 1..3$: $\beta_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, $\beta'_i \leftarrow_{\mathrm{R}} \mathbb{Z}_q$
if $(y, t) \in F$
    then for $i = 1..3$: $f_i \leftarrow g^{\beta_i}$
    else  for $i = 1..3$: $f_i \leftarrow h_{\mathrm{L}}/g^{\beta'_i}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, Q proves to P

$$\text{⋊}\, \beta'_1, \beta'_2, \beta_3 \in \mathbb{Z}_q,\ t \in S^* : \Big[g^{\beta'_1} = h_{\mathrm{L}}/f_1 \wedge g^{\beta'_2} = h_{\mathrm{L}}/f_2\Big] \vee \Big[g^{\beta_3} = f_3 \wedge (y, t) \in F\Big];$$

Note that f_1, f_2, f_3 are delivered to P via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after Q erases $\beta'_1, \beta'_2, \beta_3$.

3a. P computes:

```
if $(x, s) \in E$
    then $u_i \leftarrow f_i^{\alpha_i}$ for $i = 1..2$, erase $\alpha_2$
    else  $u_1 \leftarrow_{\mathrm{R}} \mathbb{G}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, P proves to Q

$$\text{\reflectbox{K}}\, \alpha_1, \alpha_3' \in \mathbb{Z}_q : \Big[g^{\alpha_1} = e_1 \wedge f_1^{\alpha_1} = u_1\Big] \vee \Big[g^{\alpha_3'} = h_{\mathrm{R}}/e_3\Big];$$

Note that u_1 is delivered to Q via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after P erases α_1.

3b. Q computes:

```
if $(y,t) \in F$
    then $v_i \leftarrow e_i^{\beta_i}$ for $i = 1\,..\,2$, erase $\beta_1$
    else  $v_2 \leftarrow_{\mathrm{R}} \mathbb{G}$
```

Using $\mathcal{F}_{\mathrm{ezk}}$, Q proves to P

$$\text{\reflectbox{K}}\, \beta_2, \beta_3' \in \mathbb{Z}_q : \Big[g^{\beta_2} = f_2 \wedge e_2^{\beta_2} = v_2\Big] \vee \Big[g^{\beta_3'} = h_{\mathrm{L}}/f_3\Big];$$

Note that v_2 is delivered to P via the $\mathcal{F}_{\mathrm{ezk}}$ functionality after Q erases β_2.

4a. P does the following:

```
if $(x,s) \in E$
    then if $u_2 = v_2$ then $res_{\mathrm{L}} \leftarrow 1$ else $res_{\mathrm{L}} \leftarrow 0$
    else  $res_{\mathrm{L}} \leftarrow 0$
output $res_{\mathrm{L}}$ after erasing all local data
```

4b. Q does the following:

```
if $(y,t) \in F$
    then if $u_1 = v_1$ then $res_{\mathrm{R}} \leftarrow 1$ else $res_{\mathrm{R}} \leftarrow 0$
    else  $res_{\mathrm{R}} \leftarrow 0$
output $res_{\mathrm{R}}$ after erasing all local data
```

The only communication performed here between P and Q is via secure channels and the $\mathcal{F}_{\mathrm{ezk}}$ functionality.

Theorem 7 *Under the DDH assumption for $\mathbb{G}$, protocol Π_1 realizes $\mathcal{F}^*_{\mathrm{caid}}$.*

Proof. We sketch the workings of the simulator in the case where P starts out honest and Q starts out corrupt.

In Step 2, if P is corrupted before e_1, e_2, e_3 are delivered, then we simply obtain P's input and generate the internal state as usual; otherwise, the simulator runs using α_1, α_2, and α_3', where $g^{\alpha_1} = u_1$, $g^{\alpha_2} = e_2$, and $g^{\alpha_3'} = h_{\mathrm{R}}/e_3$. These values are all that are necessary to carry out the rest of the simulation. For example, if P is corrupted between Steps 2a and 3a: then we get to see P's input s; if $s \in S$, then to show the internal state of P, we show α_1, α_2, and we show a random value in place of the α_3' we stored; if $s \notin S$, we show two random values in place of the values α_1, α_2 that we stored, and we show α_3'.

In Step 2b, we obtain the values $\beta_1', \beta_2', \beta_3, t$ from Q. If these values satisfy the second disjunct in Step 2b, then we provide t as an input to $\mathcal{F}^*_{\mathrm{caid}}$ on behalf of Q; otherwise, we provide $\perp$ as input. After obtaining the output of $\mathcal{F}^*_{\mathrm{caid}}$, we compute u_1 as $f_1^{\alpha_1}$ if the output was 1, and otherwise, choose u_1 as a random group element.

That completes the description of the simulator. The main idea of the proof that this simulation is faithful runs as follows.

Suppose the values in Step 2b satisfy the second disjunct in that step. Then Q will be forced (unless he can break the discrete logarithm problem) to provide values in Step 3b that satisfy the first disjunct in Step 3b, thus ensuring that v_2 is computed correctly. When this happens, one sees that the simulation is perfectly faithful to the protocol in this case.

Alternatively, suppose the values in Step 2b satisfy the first disjunct in that step but not the second. In this case, the simulation may not be perfectly faithful when P's input satisfies $(x, s) \in E$; however, using an argument similar to that in the proof of Theorem 4, under the DDH, the random value of u_1 shown to Q in the simulation will be indistinguishable from the correct value; moreover, under the CDH, the probability that Q could predict the correct value of u_2 is negligible, which ensures that the ideal functionality's output agrees with the protocol's output with overwhelming probability. $\square$

References

[ACP09] M. Abdalla, C. Chevalier, and D. Pointcheval. Smooth projective hashing for conditionally extractable commitments. In *Advances in Cryptology–Eurocrypt 2009*, pages 671–689, 2009.

[BCL$^+$05] B. Barak, R. Canetti, Y. Lindell, R. Pass, and T. Rabin. Secure computation without authentication. In *Advances in Cryptology–Crypto 2005*, 2005. Full version at `http://eprint.iacr.org/2007/464`.

[BH92] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology–Eurocrypt '92*, pages 307–323, 1992.

[Can05] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (December 14, 2005 version), 2005. `http://eprint.iacr.org`.

[CDPW07] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography 2007*, 2007. Full version at `http://eprint.iacr.org/2006/432`.

[CDS94] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology–Crypto '94*, pages 174–187, 1994.

[CHK$^+$05] R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology–Eurocrypt 2005*, 2005.

[CL01] J. Camenisch and L. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *Advances in Cryptology–Crypto 2001*, pages 93–118, 2001.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *34th Annual ACM Symposium on Theory of Computing*, 2002. Full version at `http://eprint.iacr.org/140`.

[CR03] R. Canetti and T. Rabin. Universal composition with joint state. In *Advances in Cryptology–Crypto 2003*, 2003. Full version at `http://eprint.iacr.org/2002/047`.

[CS97] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Advances in Cryptology–Crypto '97*, pages 410–424, 1997.

[CS03] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *Advances in Cryptology–Crypto 2003*, pages 126–144, 2003. Full version at `http://eprint.iacr.org/2002/161`.

[FO99] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Advances in Cryptology–Crypto '97*, 1999.

[GMR06] C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *Advances in Cryptology–Crypto 2006*, 2006.

[HK09] D. Hofheinz and E. Kiltz. The group of signed quadratic residues and applications. In *Advances in Cryptology–Crypto 2009*, 2009.

[JKT08] S. Jarecki, J. Kim, and G. Tsudik. Beyond secret handshakes: affiliation-hiding authenticated key agreement. In *RSA Conference, Cryptographers Track (CT-RSA 08)*, 2008.

[JL00] S. Jarecki and A. Lysyanskaya. Adaptively secure threshold cryptography: introducing concurrency, removing erasures. In *Advances in Cryptology–Eurocrypt 2000*, pages 221–242, 2000.

[MY04] P. MacKenzie and K. Yang. On simulation-sound trapdoor commitments. In *Advances in Cryptology–Eurocrypt 2004*, 2004. Full version at `http://eprint.iacr.org/2003/252`.

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