Rethinking Definitions of Security for Session Key Agreement

Wesley George and Charles Rackoff

Department of Computer Science,
University of Toronto,
Toronto, ON, M5S 3G4, Canada,
(wgeorge|rackoff)@cs.toronto.edu

March 7, 2013

Abstract

We consider session key agreement (SKA) protocols operating in a public key infrastructure, with pre-specified peers, that take no session ID as input, and output only a session key. Despite much work on SKA, we argue that there is no good definition of security for this (very natural) protocol syntax. The difficulty is that in this setting the adversary may not be able to tell which processes share a key, and thus which session keys may be revealed without trivializing the security condition.

We consider security against adversaries that control all network traffic, can register arbitrary public keys, and can retrieve session keys. We do not attempt to mitigate damage from hardware failures, such as session-state compromise, as we aim to improve our understanding of this simpler setting. We give two natural but substantially different game based definitions of security and prove that they are equivalent. Such proofs are rare for SKA. The bulk of this proof consists of showing that, for secure protocols, only compatible processes can be made to share a key. This property is very natural but surprisingly subtle. For comparison, we give a version of our definition in which processes output session IDs and we give strong theorems relating these two types of definitions.

Note: A very similar version of this paper was submitted to and rejected from TCC 2011 and TCC 2012. We had hoped to quickly create a revised version, but since we didn't, we present this version as is.

Contents

1 Introduction

The task of session key agreement (SKA) is for two parties to establish a shared random session key over an insecure network; the intuitive goal of security is that no one but the executing parties learn anything about the shared key. SKA is important because it is usually followed by a secure session protocol using the shared secret key.

There has been much work on defining security for SKA ([4, 5, 2, 16, 10, 11]). The (related) definitions of [10, 11], for example, are strong definitions of security and contain the essential ingredients of a definition of security for SKA, but they address a different syntactic notion of protocol than the original setting of [4]; the protocols of [10, 11] take "session IDs" as input. We argue that there is no satisfactory definition of security for protocols without partnering functions or session IDs, leaving a (considerable) gap for anyone trying to understand whether a protocol without these features should be considered secure.

The main contribution of this paper is to present two rather different game based definitions of security for SKA protocols without session IDs or a partnering function and prove them equivalent. This turns out to be a surprisingly difficult theorem. One feature of our definition we argue for below is that the "role bits" the processes are given not only allow for asymmetric protocols, but also serve an important security property. We also discuss a setting of SKA which has session IDs as output. Security is easier to define in this setting, and an SKA secure in this sense is also secure in our previous sense if the session IDs are ignored. In our discussion of session IDs, we explain that they can be useful sometimes in defining security but they are not necessary, and it is desirable to be able to discuss security of SKA protocols that do not use them.

1.1 Our SKA Setting, Without Session IDs

We assume a public key infrastructure[1] (PKI). Every person registered with the PKI has a *name*; associated with each name is a public key. We assume that good users choose a public key and a secret key using some fixed algorithm, while bad users may choose their public keys any way they like. The public keys might therefore not be unique, but the authority must make sure that the names are unique.[2] We will assume that everyone somehow has access to everyone else's public key; we will not discuss how.

At some point, people (with names) A and B might decide that they want to have a session, and therefore wish to exchange a session key. Typically, they will consider themselves to have different roles – for example Client and Server, and we will argue later that they *must* agree on different roles. We will abstract out these two roles by thinking of them as bits 0 or 1. How does A come to decide to launch a process intending to talk with B, in role (say) b? We will not concern ourselves with this: it happens via some insecure, and perhaps informal, discussion on the internet or elsewhere. We will say that A launches a *process* $\pi\langle A, B, b\rangle$, where π is a protocol, to denote an execution of π by A in role $b \in \{0, 1\}$, intending to share a key with B. The input to this process will be the names A and B, the bit b, the secret key of A and the public key of B. The output of a process, it it completes, will be either a session key (of the appropriate length), or a special symbol indicating failure.

Before we discuss security in the presence of an adversary, we briefly raise the following issue: if A and B launch processes to talk to each other, and no adversary is present, how do these processes actually wind up communicating? It's made more complicated by the fact that A might launch

[1] A similar development can be done for various shared private key settings, but not password-based settings.

[2] If the authority is willing to ensure that the public keys are unique, then the names would not be necessary. These issues are discussed in more detail in [15].

more than one process to speak with B at the same time. How does it all work? The short answer is: don't know, don't care. Word on the street is that it has something to do with "IP addresses" and "port numbers" and "process IDs" and a lot of other things that are very specific to particular operating systems and to particular internet protocols. All of these things are irrelevant to the logical level at which we are discussing key exchange. (Some people have suggested that having some kind of session ID as input or output might help A and B to communicate, but in fact, that would only make sense if our protocol was happening at a lower level, and if the IDs we used were specifically tailored to the hardware and software in question.) We just assume that in the absence of an adversary, A and B have no trouble communicating. The goal of security is to make sure that in the *presence* of an adversary that can completely control the internet, nothing "bad" can happen.

1.2 Why Not a Simulation Based Definition?

We have chosen a game based definition here rather than a simulation based definition such as UC (universal composability) [9]. We feel the UC setting is enormously complicated when dealing with key exchange, whereas our discussion here is self-contained, making it much easier to reason about this very subtle issue. Also the UC setting is in many respects unsuitable here. For one thing, the "ideal world" model is arbitrary in many respects. Another problem with UC is that matching "session IDs" are usually assumed as input to the processes. This makes sense when defining certain kinds of cryptographic primitives (for example, a kind of bit commitment), but it is not appropriate when establishing something like an "authenticated channel" is part of what the protocol is supposed to achieve, not assume to begin with. We could some insecure communication has happened before the protocol begins, establishing proto session IDs, which are then checked as part of the protocol, however this would constrain protocols to be of a particular (unnatural) form, and would not allow us to define what security means for more general kinds of protocols. Actually, if we are going to insist on session IDs, it makes more sense to have them as outputs; this is a convenience for definitions, but again it unnecessarily constrains the protocols we consider. Session IDs are discussed more below.

It makes sense to ask if there is actually something *wrong* with the definition of SKA security presented in this paper. Is there some reason it cannot be used for the typical intended purposes? We discuss below that there are additional types of "secondary" security properties that might be desirable, but these are not necessarily covered by a UC style of definition either. In fact, our definitions raise the question of whether there exists an exactly equivalent definition in a simulation style, without changing the syntax of SKA.

1.3 Defining Security, a Game Based Definition

An adversary chooses names for good guys (or honest parties) as well as names of bad guys; bad guys are just different versions of himself, and he chooses their public keys however he likes. He can launch processes of the form $\pi\langle A, M, b\rangle$, where A is honest and M may be good or bad, and he can read from and write to these processes.

Intuitively, an SKA protocol π is secure if no one other than honest parties A or B can learn anything about the session key output by a $\pi\langle A, B, b\rangle$ process. We model this by asking whether an adversary can distinguish the key output by the protocol from a key chosen uniformly at random. However, it is important to note that "unrelated" protocol invocations must generate independently random session keys. We model this by saying that the adversary should be unable to distinguish the challenged session key from random even when he is allowed to open (or reveal) other session

keys. However, we have to restrict this adversary in some way, for the following reason. We say two processes are compatible if they are of the forms $\langle A, B, b\rangle$ and $\langle B, A, \bar{b}\rangle$. An adversary might create two compatible processes, and have them properly communicate with each other, so that they will output the same session key; if we allow him to challenge one of the processes and open the other, he will be able to break every protocol. So how should we constrain the adversary?

We say two processes are *partners* if they are compatible and output the same session key. One class of definitions allow the adversary to somehow know when processes are partners. One way to do this [4, 5, 8] is to assume that whether two completed compatible processes share a key is efficiently computable from the network transcript. This will not generally be the case, so we would have to restrict ourselves to certain kinds of protocols. For example, the "matching conversations" definition of [4] makes processes partners if every message generated by one process was delivered to the other and vice-versa. While this definition suffices for the analysis of certain protocols, it is unacceptable in the general case as any protocol can be made insecure by adding an extra bit to a message where the bit is supposed to be ignored.

One of our definitions in this paper, "partner security", gives the adversary an oracle which tells him which of the compatible pairs are partners. This seems to us to be the cleanest form of this kind of definition.

Another way of making it easy to tell if two processes are partners is to change the syntax of what we mean by SKA. The cleanest way to do this is have processes output a "session ID" in addition to a session key, where the adversary automatically sees all the session IDs, and two compatible processes are considered as partners if they output the same session ID. In order for this to make sense, the protocol must also satisfy a "session ID security condition" where the adversary is not able to make two compatible processes output the same session ID unless they have also output the same session key. [3, 13, 12] are examples of protocols that output a session ID. In this paper we formalize security for SKA with session ID as output. Furthermore, we prove that if the session IDs are simply dropped, the result is a secure SKA protocol in our standard syntax, according to our "partner security" definition. We also show how to "convert" any secure SKA protocol in the standard syntax to one in the syntax with session ID as output, but this involves changing and adding to the protocol. This is why we prefer a definition in the standard syntax without session IDs. We want to understand what it means for such a protocol to be secure.

We lastly discuss one way of defining SKA security in the standard syntax without making the adversary able to compute partnering. If the adversary winds up opening a partner of the challenged process (or challenging a partner of an opened process), then we *penalize* him. This is a very tricky thing to do right, and one might wind up making the definition too strong or too weak. The Master's thesis of Arruda [1] discusses some versions of this style of definition that were used in some early course notes of Rackoff. In this paper, we present a new version of this style of definition that we call "penalty security", and we prove that it is equivalent to partner security.

1.4 Role-Bit Security, and other Kinds of Security

Our security definitions for SKA presented here implicitly associate a security condition with the role bits. In particular, it is a (highly nontrivial) consequence of our security definitions that an adversary will not be able to make two processes share a key if they have the same role bit. In particular, a process of type $\langle A, B, b\rangle$ should not be tricked into sharing a key with a process of type $\langle B, A, b\rangle$. For example, if the roles are interpreted as Client and Server, an adversary should not be able to make two Clients share a key, or make two Servers share a key.

Why do we find this security condition desirable? For one thing, it is easy to achieve. Every SKA protocol we know of already has this property, or can be easily made to have this property by

signing one extra bit in some of the flows. The most important reason to have role-bit security is because it is natural for people designing a session protocol that uses a shared session key to *assume* that the parties start with an asymmetry. For example, often session protocol designers assume that the party that "initiated" the key exchange will begin the session; but this only makes sense if the parties agree in a secure way on who initiated the exchange, and this is essentially the purpose in having a secure role bit. For another example, if session protocol designers think of one party as a Server and the other as a Client, they would be horrified if it were possible for two Servers or for two Clients to share a key. Or often one specifies that a shared session key is expanded into two new keys, one of which is to be used for communication in one direction, and the other of which is to be used for communication in the other direction. But which party gets which of the two keys? The danger of having a session where there is no initial (secure) asymmetry between the two parties is that an otherwise secure protocol can become insecure. For example, both parties may use the same string (derived from the shared key) as a one-time pad. Of course, there are protocols for securely establishing a session asymmetry if none initially exists, but it is more common to assume such an asymmetry exists as a result of the key exchange. Sometimes it is suggested that the names of the parties be used to supply an asymmetry, with the (lexicographically) lower name getting (for example) role 0. This doesn't work, of course, if the two names are the same. But it is a bad idea even if the two names are different. Since the the designer of the session protocol should not have to know how the session key was exchanged, the session protocol should no longer have access to these names; they should be discarded as existing on a lower logical level from the intent of a person who has shared a session key and now wishes to engage in a session. In fact, a secure session can take place in a setting where no names exist. For example, the two parties might have gotten together and flipped some coins to create a session key, or they might have used a long-term shared private key to exchange a session key.

Some SKA security definitions in the literature do insist on role-bit security, whereas some others do not. For example, [10, 11] does not have role-bit security while [9] does. Of course, if for some reason one wants to avoid insisting on role-bit security, one can always adjust our definitions. The obvious adjustment would be to change the definition of what it means for two processes to be "compatible", so that $\langle A, B, b\rangle$ would be compatible with $\langle B, A, b'\rangle$, even if $b = b'$. This would by itself not be adequate, however, since it allows for the following bad situation: imagine that every honest party A has a special string K_A as part of its secret key. Then we could let every process $\langle A, A, b\rangle$ output the same session key K_A, and this would be consistent with this revised definition of security[3]. In order to avoid this, we would also have to add to the definition the rather unnatural stipulation that an adversary should not be able to make three processes output the same session key.

There are certain other properties that are consequences of our definitions. For example, although we insist that the adversary decides at the moment a new name is chosen whether that person is good or bad, it would not give him additional power if we allowed him to delay this decision. This is a useful kind of security against "dynamic corruption". Note however, it does not yield the very fine-grained type of dynamic corruption that some UC definitions do.

There are additional security properties for SKA protocols that one might be interested in. Most interesting are the "secondary" security properties, where we wish to retain some security even when an adversary has obtained access to keys or other material we normally assume physically secure, perhaps by breaking into some computer. Examples are "forward security" and "key compromise impersonation resilience", or security in the face of "session state compromise". These properties

[3]This is because if an adversary challenges one of these processes, it would not be allowed to open any of the others.

are not implied by our definition as we have aimed, with this work, to improve our understanding of this simpler setting.

1.5 Our Contributions

We give two equivalent game based definitions of security for SKA that resolve the difficulty of appropriately restricting the adversary in a novel way.

The adversary challenges a completed process, and is given either the session key or a random string, and he tries to guess which. He is also allowed open other completed processes. For our first definition, which we dub "partner-security", we simply allow the adversary to ask if two completed compatible processes share a key; if so, he is not allowed to open one and challenge the other, in either order. For our second definition, dubbed "penalty-security", we do not restrict what processes may be opened or challenged but instead we "penalize" the adversary for cheating: if the adversary challenges a process sharing a key with an open compatible process (i.e. challenges a process partnering with an open process), we ignore the run by replacing the adversary's output (i.e. its guess concerning the challenge key) with a random bit (making the adversary win with probability exactly half); if the adversary opens a partner of the challenged process, he receives the challenge key. Both these definitions are simpler than the others we have seen, except for some which gain simplicity by restricting the protocols that can be considered (for example, by insisting on session IDs or computable partnering functions).

Our main technical contribution is proving that these two definitions are equivalent. While partner-security easily implies penalty-security, the converse is not so direct. The difficulty is in showing that penalty-security implies that an adversary cannot (except with negligible probability) make incompatible processes share a session key. The hard case here is the following. Say that the adversary constructs two processes of the form $\langle B, A, 1\rangle$; we want to prove that he cannot make them both output the same key, k. At first this seems easy: if the adversary could do this, then he could challenge one of the $\langle B, A, 1\rangle$ processes and open the other one. The problem is that he might already have opened a $\langle A, B, 0\rangle$ process that also outputs k (and in fact this might be necessary in order to make the other two processes output k), causing him to pay the penalty of outputting a random bit.

We also give a definition of secure SKA with a syntax that has session IDs as outputs (that the adversary sees) as well as session keys. We show that a protocol that is secure with this syntax according to the new definition remains secure according to the old definition when the session IDs are omitted. We also show how to convert a protocol with session IDs into one without session IDs, preserving security.

1.6 Related Work

Much work has been done on SKA, some of which has already been surveyed. The first definition of security for SKA, and the basis for much later work, was given by Bellare and Rogaway [4] for the setting where each pair of parties shares a long-term private key, and this was adapted to the setting of public key infrastructure by Menezes, Blake-Wilson and Johnson [6, 7]. These papers use the notion of "matching conversations" to define which processes are partners (and appropriately constrain the adversary).

[5] introduced the first explicit notion of partnering function in the setting of server-mediated key-exchange[4]. [14] later showed that the function does not work as intended. Partnering functions

[4]This is also known as the Kerberos setting: all parties share a key with a server that is always online.

are also discussed in the simulation based definition of [16]. The notion of session IDs as output for partnering first appeared in [2], and subsequently in many works (e.g. [3, 13, 12, 8]).

Protocols with session IDs as input were first considered in [10]. Here they give a game based definition and prove it sufficient for a given notion of secure session. This was later shown in [11] to be equivalent to a UC definition of SKA with static corruption. This is the only example we know of where two substantially different definitions of security for SKA are shown to be equivalent.

Organization

We begin, in Section 2, by giving our definitions. In Section 3 we prove that our two definitions of security are equivalent. Section 4 relates our definitions of security to those involving session IDs.

2 Definitions

2.1 Protocol Syntax

Definition 2.1. A *session key agreement protocol*, is a pair of polytime ITMs (Gen, π) satisfying the following conditions:

- Gen on input 1^n outputs a pair of keys (pub, pri).
- If an instantiation of π terminates, it either outputs an n-bit string (the intended session key) or `fail`.
- Let $(pub_1, pri_1), (pub_2, pri_2)$ be in the range of $Gen(1^n)$, $b \in \{0,1\}$. Let $N_1, N_2 \in \{0,1\}^n$; we think of these as the names of the participants. If the instantiations of π with inputs

$$P_1 = \pi(1^n, N_1, pri_1, N_2, pub_2, b)$$
$$P_2 = \pi(1^n, N_2, pri_2, N_1, pub_1, \overline{b})$$

  are allowed to compute such that every message generated by one is sent to the other and vice-versa, both processes terminate and output the same session key.

We refer to the SKA protocol (Gen, π) just as π. For fixed n and when it is understood that A is the name of an honest party with private key pri_A and B is the name of a party with public key pub_B, we will write $\pi\langle A, B, b\rangle$ to mean an instantiation of π on inputs 1^n, A, pri_A, B, pub_B and b. We call such instantiation a process. When there is no danger of confusion about what protocol we mean, we will write $\langle A, B, b\rangle$ instead of $\pi\langle A, B, b\rangle$.

If two processes output the same key, we say that they have *shared* a key. Of course, arbitrary pairs of processes should not share keys.

Definition 2.2. Two processes $\langle A, B, b\rangle$ and $\langle C, D, b'\rangle$ are said to be *compatible* if $A = D$, $B = C$, $b' = \overline{b} = 1 - b$. e.g. $\langle A, B, 0\rangle$ and $\langle B, A, 1\rangle$ are compatible processes.

When two compatible processes share a key, we call them *partners*. The definition of compatible plays an important role in our definition of security. It should be viewed as a bad thing if an adversary can arrange for non-compatible processes to share a key. While this is not an explicit requirement of our security definition, we will show in Section 3 that such protocols are insecure. This is a natural property to ask of a security definition, but it is difficult and subtle proof. An important corollary is that (except with negligible probability) for secure protocols, at most two processes can share a key and only if they are compatible. This corollary depends on the fact that

the role-bit is included in the definition of compatible; were the role-bit absent from Definition 2.2 our definitions of security would name protocols secure even though we could arrange for three or more processes to share a key. It is interesting to note that even though the definitions of [10, 11] permit processes with the same role-bit to share a key, they *do not* permit three processes to share a key. This latter feature is a consequence of the way those definitions treat session IDs as inputs.

2.2 Security

We begin by formalizing how the network of processes may be controlled by the adversary. Our security experiments are formulated as an interaction between an adversary and the "ring-master" who simulates a network of protocol executions by honest players (i.e. players attacked by the adversary). The ring-master keeps the appropriate private keys secret from the adversary and also administrates the security experiment.

Definition 2.3. The *template ring-master*, denoted $\mathcal{RM}^T$, is an ITM that, on input 1^n, and an SKA protocol (Gen, π), that accepts the following queries:

- `register`(A): if $A \in \{0,1\}^n$ is not the name of any registered player, run $Gen(1^n)$ to get a new key pair (pri, pub), record A as the name of the honest player with key pair (pri, pub) and return pub.
- `register`(M, pub): if $M \in \{0,1\}^n$ is not the name of any registered player, record $M \in \{0,1\}^n$ as the name of the dishonest player with public key pub.
- `initialize`(A, B, b): If A is the name of an honest player and B is the name of a (possibly dishonest) player, start simulating π on (A, pri_A, B, pub_B, b). We associate an index $i \in \mathbb{N}$ with this process and subsequently refer to this process as $\langle A, B, b\rangle_i$. Return i to the adversary along with any initial message generated by the process.
- `send`(A, B, b, i, m): simulate $\langle A, B, b\rangle_i$ as having received message m; return any message m' generated by $\langle A, B, b\rangle_i$, as well as report if the process terminates or fails.
- `challenge`(A, B, b, i): If $P = \langle A, B, b\rangle_i$ is a terminated process that has not been challenged, both A and B are honest players, and no other process has been challenged, do the following: pick and output $b^* \in_R \{0,1\}$ then report k^* where k^* is P's key if $b^* = 0$ and a uniformly random element of $\{0,1\}^n$ otherwise.
- `open`(A, B, b, i): If $P = \langle A, B, b\rangle_i$ is a terminated process that has not been challenged, return P's session key; otherwise return nothing.

When the parameters in question are clear, we'll write $\mathcal{RM}^T(1^n, Gen, \pi)$ as $\mathcal{RM}^T_\pi$ or simply $\mathcal{RM}^T$ (as well as for the subsequent modified versions of $\mathcal{RM}^T$). We will refer to the challenged process as P_{CH} and to its session key as k_{CH}. Note that if $b^* = 0$, $k_{CH} = k^*$ while if $b^* = 1$, then $k_{CH} \neq k^*$ except with negligible probability.

Note that $\mathcal{RM}^T$ only simulates processes executed by honest players. Also note that though the ring-master assigns an index to each process, this is only used by the adversary to direct the ring-master in coordinating the network; this index is *not* available to the process in any form.

2.2.1 Partner-Security

Our first definition of security matches the spirit of those involving session IDs: the ring-master will tell the adversary if two compatible processes share a session key. $\mathcal{RM}^P$ will not accept challenges to partners of opened processes or reveal the session key of partners of the challenged process.

Definition 2.4. The *ring-master for the partner-game*, denoted $\mathcal{RM}^P$ is an ITM that takes as input a security parameter 1^n and an SKA protocol (Gen, π) and behaves as $\mathcal{RM}^T$ but with the following additional query:

- `partner`(A, B, b, i, j). If processes $\langle A, B, b\rangle_i$ and $\langle B, A, \overline{b}\rangle_j$ have both terminated, return true if they have output the same session key and false otherwise.

and with the modifications to `open`, `challenge` as follows:

- `open`(A, B, b, i): return P's key as usual *unless* P is, or partners with, the challenged process.
- `challenge`(A, B, b, i): Administrate the challenge as usual *unless* P is, or partners with, an opened process.

We note that partnering queries concerning P_{CH} are answered based on k_{CH}, the key output by P_{CH}, and *not* on k^*, the challenge key. In particular when $k^* \neq k_{CH}$ (as usual when $b^* = 1$), P and P_{CH} are reported as partners if and only if P's session key is k_{CH}. Also note that we could give the adversary more information (potentially making her stronger) by allowing her to ask if *any* pair of processes share a key (rather than any pair of *compatible* processes). Adding such information does not increase the power of the adversary (proof to appear in the full version) so we prefer the simpler formulation above.

The *partner-security game played by* $\mathcal{M}$ *against* π is the joint-computation of $\mathcal{M}$ and $\mathcal{RM}^P(\pi)$; the game ends when $\mathcal{M}$ outputs a bit (her guess at b^*) and halts. If $\mathcal{M}$ terminates without outputting a bit, $\mathcal{M}$'s output is taken to be a random bit; if $\mathcal{M}$ terminates without challenging a process (and thus without $\mathcal{RM}^P_\pi$ having chosen b^*), b^* is also taken to be a random bit. Let $W^P_\pi(\mathcal{M}, 1^n)$ denote the event that $\mathcal{M}$ correctly guesses b^*.

Definition 2.5. We say that π *partner-secure* if for all probabilistic polynomial time adversaries $\mathcal{M}$ there is a negligible function ϵ such that

$$\Pr[W^P_\pi(\mathcal{M}, 1^n)] < 1/2 + \epsilon(n) \tag{1}$$

2.2.2 Penalty-Security

For our second definition, rather than restrict any of the adversary's actions, we allow the adversary to open or challenge any process, but we may make some changes to the simulation to prevent trivial session key compromises.

Definition 2.6. The *ring-master for the penalty-security game*, denoted $\mathcal{RM}$, is an ITM that on input of 1^n and an SKA protocol (Gen, π) behaves as $\mathcal{RM}^T$ but with the following modifications:

- `open`(A, B, b, i):
  - If $\langle A, B, b\rangle_i$ partners with the challenged process, return k^*. (Recall that k^* is the challenge key which the adversary must identify as either a random key or the real key).
  - Otherwise, return P's key as usual.
- `challenge`(A, B, b, i): if $\langle A, B, b\rangle_i$ is or partners with an open process, pick and output $b^* \in_R \{0, 1\}$ and halt[5].

[5] As $\mathcal{M}$ receives no information about b^*, her guess will be correct with probability exactly 1/2.

The *penalty game played by an adversary $\mathcal{M}$ against an SKA protocol π* is the joint computation of $\mathcal{M}$ and $\mathcal{RM}_\pi$. The game ends when $\mathcal{M}$ outputs a bit b(again, $\mathcal{M}$'s guess for b^*) and halts. If $\mathcal{M}$ terminates without challenging a process and therefore $\mathcal{RM}_\pi$ has not output a bit, b^* is taken to be a random bit. Let $W_\pi(\mathcal{M}, 1^n)$ denote the event that $\mathcal{M}$ correctly guesses b^*.

Definition 2.7. π is *penalty-secure* if for all probabilistic polytime adversaries $\mathcal{M}$ there exists a negligible function ϵ such that

$$\Pr[W_\pi(\mathcal{M})] < 1/2 + \epsilon$$

Other definitions of security consider adversaries that may "corrupt" honest parties; we think of corruption as giving the adversary the long-term private key and past randomness used by a party. We do not equip our adversaries with such ability as it adds no extra power in this very simple setting. Ultimately, if a $\langle A, B, b\rangle$ process is challenged, the adversary must not corrupt A or B. On account of this, an adversary that is not allowed to corrupt parties can easily simulate an adversary that can by guessing which process will be challenged.

2.3 An Example Protocol

Thus far we have argued that our definitions capture a strong notion of security. Of course definitions that rule all protocols as insecure have similar properties, so it is important to show that our definitions are non-trivial. In Appendix A we present a standard key transport protocol and proof of security.

3 Equivalence of our Definitions

We aim to prove the following theorem.

Theorem 3.1. *Let (Gen, π) be an SKA protocol.*

$$\pi \textit{ is partner-secure} \iff \pi \textit{ is penalty-secure}$$

The forward implication is trivial as an adversary playing the partner-security game can easily simulate the ring-master for the penalty-security game. The reverse implication requires an adversary playing the penalty-security game simulate knowledge about which processes partner. We will need two properties concerning penalty-security. We will say a $\langle A, B, b\rangle$ process is totally honest if both A and B are honest players. The first is somewhat specialized and easy to prove.

Lemma 3.2. *Let (Gen, π) be an SKA protocol, $\mathcal{M}$ an adversary. Let $E_\pi(\mathcal{M})$ be the event that when $\mathcal{M}$ plays the penalty-security game against π that $b^* = 1$, and there exists a totally honest process who outputs k^* as its key.*

$$\pi \textit{ is penalty-secure} \implies \forall \mathcal{M} \Pr[E_\pi(\mathcal{M})] \textit{ is negligible}$$

Recall that $b^* = 1$ means that k^* is chosen uniformly at random and so k^* is not P_{CH}'s key (except with negligible probability). Lemma 3.2 is easily proved. The second property is of independent interest, and is natural to ask of a definition of security.

Theorem 3.3. *Let (Gen, π) be an SKA protocol. Let $\mathcal{M}$ be an adversary that opens every process as soon as it terminates and does not challenge any process; let $M_\pi(\mathcal{M})$ be the event (over network arrangements by $\mathcal{M}$) that two non-compatible totally honest π processes share a key.*

$$\pi \textit{ is penalty-secure} \implies \forall \mathcal{M} \Pr[M_\pi(\mathcal{M})] \textit{ is negligible}$$

A corollary of interest is is that at most two totally honest processes ever output the same key, and only if they are partners. From here, the proof of Theorem 3.1 is simple thanks to the mechanics of the penalty-game: the partner of the challenged process is apparent as it is the only process that outputs k^*, so simulating the partnering oracle is easy. See Appendix C for a complete proof. Given Theorem 3.1, we subsequently refer to SKA protocol meeting Definitions 2.5 and 2.7 simply as 'secure'.

Despite the naturalness of the statement Theorem 3.3, the proof is subtle and difficult. Suppose $\mathcal{M}_M$ arranges for non-compatible processes P_a and P_b to share a key. If we are guaranteed that P_b has no earlier partner, then breaking π is simple: simulate $\mathcal{M}_M$, open P_a, challenge P_b, and answer the challenge in the obvious way; we are not penalized for this strategy since P_a and P_b are (crucially) non-compatible processes. The difficult case is when P_a and P_b are of the same type (e.g. $\langle B, A, 1\rangle$) and there is a third process P_m of compatible type (e.g. $\langle A, B, 0\rangle$) partnering with and terminating before both P_a and P_b, and needs to be opened in order to engineer the mismatch; in this case neither P_a nor P_b can be challenged. We give a detailed proof in Appendix B. Of course a definition of security could be augmented to rule out such protocols as insecure; it is a nice fact that this undesirable situation is already ruled out as a consequence of the existing definition.

4 Relating to Protocols with Session IDs

There are two standard ways that session IDs are incorporated into a protocol: they are either taken as input as in [10], or they are provided as output as, effectively, in [13] and as formalized in [12]. We find the later notion more natural as explained in the introduction.

4.1 Definitions

Definition 4.1. A *session key and id agreement protocol* (SKAwSID protocol), (Gen, θ), is a pair of algorithms satisfying the following conditions:

- Gen on input 1^n outputs a pair of keys (pub, pri).
- If an instantiation of θ terminates then it outputs either a pair of n-bit strings, called a *session key* and a *session ID*, or `fail`.
- Let $(pub_1, pri_1), (pub_2, pri_2)$ be in the range of $Gen(1^n)$, $b \in \{0,1\}$ and $N_1, N_2 \in \{0,1\}^n$. If the instantiations of θ with inputs

$$P_1 = \theta(1^n, N_1, pri_1, N_2, pub_2, b)$$
$$P_2 = \theta(1^n, N_2, pri_2, N_1, pub_1, \bar{b})$$

  are allowed to compute such that every message generated by one is sent to the other and vice-versa, both processes terminate and output the same session key and session ID.

We use the same conventions when referring to SKAwSID protocols as with SKA protocols (e.g. refer to (Gen, θ) simply as θ, use $\theta\langle A, B, b\rangle$ to mean an invocation of θ on $(1^n, A, pri_A, B, pub_B, b)$, etc.). We call two compatible processes $\theta\langle A, B, b\rangle$ and $\theta\langle B, A, \bar{b}\rangle$ *partners by session ID* if they output the same session ID.

session keys output by SKAwSID protocols should have the same security properties as session keys output by SKA protocols. The session ID should give the adversary an explicit way of identifying partners in the network and thus which processes may or may not be opened or challenged.

However we also need to ensure that arbitrary restrictions are not imposed on the adversary (e.g. if every process outputs the same session ID we've severely restricted the adversary without providing any useful information), so we consider an SKAwSID protocol insecure if an adversary can arrange for two processes to output the same session ID but different session keys.

Definition 4.2. The *ring-master for the SKAwSID security game*, denoted $\mathcal{RM}^{SID}$ is an ITM that takes a security parameter 1^n and an SKAwSID protocol (Gen, θ) as input and behaves as $\mathcal{RM}^T$ with the following modifications:

- `send`(A, B, b, i, m): if $\langle A, B, b\rangle_i$ terminates after receiving m, return its session ID (in addition to any message generated).
- `open`(A, B, b, i): if $\langle A, B, b\rangle_i$ has terminated and is not, nor partners by session ID with the challenged process, return $\langle A, B, b\rangle_i$'s session key.
- `challenge`(A, B, b, i): If $\langle A, B, b\rangle_i$ has terminated and is not, nor partners by session ID with an opened process, administrate the challenge as usual.

For an adversary $\mathcal{M}$, the *session ID security game played by $\mathcal{M}$ against θ* is the joint computation of $\mathcal{M}$ and $\mathcal{RM}^{SID}(\theta)$. We say that $\mathcal{M}$ *breaks the session ID* if two compatible process output the same session ID but different session keys and write $W_\theta^{SID}(\mathcal{M})$ to denote the event this occurs. We'll write $W_\theta(\mathcal{M})$ to denote the event that $\mathcal{M}$ guesses the challenge bit correctly. θ is said to be secure if for all probabilistic polynomial time adversaries $\mathcal{M}$ there is a negligible function ϵ such that

$$\Pr[W_\theta(\mathcal{M})] < 1/2 + \epsilon$$
$$\Pr[W_\theta^{SID}(\mathcal{M})] < \epsilon$$

Definition 4.2 is essentially the definition of [13] without a "session state reveal" query. We note the following easy lemma:

Lemma 4.3. *For a SKAwSID protocol θ and adversary $\mathcal{M}$, let $M_{\mathcal{M},\theta}$ be the event that, during the joint-computation of $\mathcal{M}$ and $\mathcal{RM}_\theta^{SID}$ that two processes share a session key but output different session IDs. If θ is secure (in the sense of Definition 4.2) then*

$$\forall \mathcal{M} \ \Pr[M_{\mathcal{M},\theta}] \text{ is negligible}$$

4.2 Transformations

Since SKAwSID protocols are syntactically different from our SKA protocols, we cannot show directly compare them Instead, we give a simple transformations from SKA protocols to SKAwSID protocols and vice-versa that preserve security.

4.2.1 From Secure SKAwSID to Secure SKA

Let θ be a SKAwSID protocol, we define a corresponding SKA protocol π_θ which behaves as θ with the exception that when θ would have terminated outputting both session key k and session ID s, π_θ outputs only k.

Theorem 4.4. *If θ is a secure SKAwSID protocol, then π_θ is a secure SKA protocol.*

Proof. Since θ is fixed, we will refer to π_θ simply as π. Let $\mathcal{M}_\pi$ be an adversary breaking the partner-security of π. Let $\mathcal{M}_\theta$ be the following adversary attacking the SID-security of θ that simulates $\mathcal{M}_\pi$, creating θ processes in place of π processes, doesn't report the session IDs to $\mathcal{M}_\pi$ but when asked if two compatible processes share a session key, answers yes if and only if their session IDs match.

$\mathcal{M}_\theta$ simulates $\mathcal{M}_\pi$ so long as two θ processes share a session key if and only if they share a session ID. But if the later is false, then θ is insecure either by definition or by Lemma 4.3. □

4.2.2 From Secure SKA to Secure SKAwSID

Let π be an SKA protocol. We define a corresponding SKAwSID protocol θ_π which, operating in the same infrastructure as π, behaves as follows: θ_π on security parameter n, runs π on security parameter $2n$; when π terminates outputting a $2n$-bit session key, θ_π outputs the first half as the session key, and the second half as the session ID.

Theorem 4.5. *If π is a secure SKA protocol then θ_π is a secure SKAwSID protocol.*

The proof is a straightforward hybrid argument and will appear in the full version of the paper.

References

[1] Nelson Arruda. Formal definitions of key exchange models and the effects of restrictions on adversaries. Master's thesis, University of Toronto, 2005. `http://proquest.umi.com/pqdlink?did=974469881&Fmt=7&clientId=12520&RQT=309&VName=PQD`.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. A modular approach to the design and analysis of authentication and key exchange protocols (extended abstract). In *STOC*, pages 419–428, 1998.

[3] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attack. In *EUROCRYPT*, pages 139–155, 2000.

[4] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *CRYPTO*, pages 232–249, 1993.

[5] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: the three party case. In *STOC*, pages 57–66, 1995.

[6] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. Key agreement protocols and their security analysis. In *IMA International Conference on Cryptography and Coding*, pages 30–45, 1997.

[7] Simon Blake-Wilson and Alfred Menezes. Entity authentication and authenticated key transport protocols employing asymmetric techniques. In *Security Protocols Workshop*, pages 137–158, 1997.

[8] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Steven C. Williams. Composability of bellare-rogaway key exchange protocols. In *CCS*, 2011.

[9] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 v20051214:064128, December 2005. `http://eprint.iacr.org/`.

[10] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *EUROCRYPT*, pages 453–474, 2001.

[11] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In *EUROCRYPT*, pages 337–351, 2002.

[12] Vladimir Kolesnikov and Charles Rackoff. Key exchange using passwords and long keys. In *TCC*, pages 100–119, 2006.

[13] Hugo Krawczyk. Hmqv: A high-performance secure diffie-hellman protocol. In *CRYPTO*, pages 546–566, 2005.

[14] Kim kwang Raymond Choo, Colin Boyd, Yvonne Hitchcock, and Greg Maitland. On session identifiers in provably secure protocols - the bellare-rogaway three-party key distribution protocol revisited. In *Conference on Security in Communication Networks (SCN)*, pages 351–366, 2004.

[15] Charles Rackoff. On "identities", "names", "names", "roles" and security: A manifesto. Cryptology ePrint Archive, Report 2011/214, 2011. `http://eprint.iacr.org/`.

[16] Victor Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999. `http://www.shoup.net/papers`.

Appendices

A An example of a secure SKA protocol

Let $\Pi_S = (Gen_S, Sign, Ver)$ be a public-key signature scheme and $\Pi_E = (Gen_E, Enc, Dec)$ be a public-key encryption scheme; we suppose that for security parameter n, Π_E encrypts $2n$ bit strings. We define protocol π_T as follows:

Protocol π_T:

- *Gen*: on input (1^n), run $Gen_S(1^n)$ and $Gen_E(1^n)$ to get $(sign, ver)$ and (enc, dec); return $(pub = (ver, enc), pri = (sign, dec))$.
- $\pi_T\langle A, B, 0\rangle$:
  1. Pick $r \in_R \{0,1\}^n$. Send r.
  2. Upon receipt of α, σ, if $Ver_B(\alpha r, \sigma) \neq 1$, fail. Otherwise let $(k, N) = Dec_A(\alpha)$. If $N \neq B$, fail. Otherwise output k and terminate
- $\pi_T\langle C, D, 1\rangle$: Upon receipt of $r \in \{0,1\}^n$, pick $k \in_R \{0,1\}^n$. Compute $\alpha \leftarrow Enc_D(kC)$, $\sigma \leftarrow Sign_C(\alpha r)$. Send $\alpha\sigma$ and halt, outputting k.

Proposition A.1. *If Π_S is a secure public-key signature scheme and Π_E is a CCA2-secure public-key encryption scheme then π_T is a secure SKA protocol.*

Proof. Given the equivalence of our two definitions of security, it suffices to show how to use an adversary breaking either the penalty-security or the partner-security of π_T to compromise one of our assumptions. We choose to show how an adversary breaking the penalty-security since this definition is more novel. So suppose $\mathcal{M}$ is an adversary that breaks the penalty-security of π_T.

Let E_S be the event that, during the security game of $\mathcal{M}$ against π_T, the adversary forges a message. More specifically, let E_S be the event that there exists an i and honest players A,B, such that $\langle A, B, 0\rangle_i$ sent r as its first message and received $\alpha\sigma$, $Ver_B(\alpha r, \sigma) = 1$, and no $\langle B, M, 1\rangle$ process received r and sent $\alpha\sigma'$ for some σ' and M.

If should be clear that if $\Pr[E_S]$ is significant, then we can arrange to break the security of the signature scheme by simulating $\mathcal{M}$ playing the penalty-security game. So suppose otherwise. We will argue that $\mathcal{M}$ is effectively breaking the (CCA2-)security of Π_E, and that we can exploit this fact to simulate $\mathcal{M}$ and break Π_E ourselves.

Let P_0 be a role-0 process that sent r as its first message and receives $\alpha\sigma$. Now if P_0 terminates and outputs a session-key (rather than failing), we know that σ is a valid signature over αr. Since $\Pr[E_S]$ is negligible, we know that there is a role-1 process P_1 that received r as its first message and output $\alpha\sigma$, and these two processes are partners. Now P_1 chose it session-key uniformly at random, so we know that, except with negligible probability, no other role-1 process will output the same session-key, so P_1 is P_0's unique partner. Further, a role-0 process will only accept the $\alpha\sigma$ sent by P_1 if this process sent r in the first round; but the first message sent by a role-0 process is chosen uniformly at random, so if no signature's are forged, then P_0 is P_1's unique partner.

Given the preceding discussion, we suppose that in any play of the penalty-security game by $\mathcal{M}$, there is a unique role-1 process that is either challenged or partners with the challenged process. Suppose that process is $\langle B, A, 1\rangle_i$, and that we know this fact. We can break Π_E on A's key pair as follows: we generate a signing and encryption key pair for each party, with the exception that rather than generating an encryption key pair for A, we will use the inputted public key *pub* as A's public key (and we do not know A's private key); our challenge messages will be k_0B and k_1B for independent and uniformly chosen k_0, k_1; when $\mathcal{M}$ sends $\langle B, A, 1\rangle_i$ a message r, we respond with $\alpha^*\sigma$ where α^* is the challenge ciphertext (i.e. either an encryption of k_0B or k_1B under A's private key) and σ is B's signature of α^*r; when $\mathcal{M}$ challenges $\langle B, A, 1\rangle_i$ or its partner, we give k_0 as the challenge key and continue simulating $\mathcal{M}$ to completion; since we have argued that we know which processes are partners, it is easy to recognize if $\mathcal{M}$ is cheating and we can flip a coin (as our guess for the message encrypted by the challenge ciphertext) when this happens; most processes are easy to simulate as we have the private keys of all parties - the difficulty is in simulating $\langle A, M, 0\rangle$ processes (for dishonest players M) which requires A's decryption key, however since we have access to the decryption oracle for A's decryption key, we can simulate these processes as well; if $\mathcal{M}$ guesses that k_0 was the real key, we guess that α^* is an encryption of k_0B, whereas if $\mathcal{M}$ guesses that k_0 is a random key, we guess that α^* is an encryption of k_1B. It should be easy to see that this simulation correctly guesses the plaintext of α^* whenever $\mathcal{M}$ wins the penalty game.

This sketch suppresses many details, but is the essence of the attack on Π_E. We don't know the names A,B or the process number i, but we can simply guess these at the start of the run. Our guesses will be correct with significant probability[6]. If our guess is wrong, this fact will be revealed when $\mathcal{M}$ tries to open or challenges a process other than $\langle B, A, 1\rangle_i$ or its partner (if it exists), and we 'abort' and guess randomly about the challenge ciphertext. Since we win with probability exactly half whenever our guesses are wrong, and with significant probability, win with the same probability that $\mathcal{M}$ does against π. Combining these facts gives us that the outlined adversary breaks the CCA2-security of Π_E. □

[6]Clearly i comes from a set of polynomial size. We guess the names A and B by guessing their indices in the list of honest players, which also has polynomial size.

B Proof of Theorem 3.3

Suppose π is an SKA protocol and that an adversary $\mathcal{M}_M$ can arrange, with significant probability, for two totally honest non-compatible π processes to share a key. Recall that we suppose, without loss of generality, that $\mathcal{M}_M$ opens every terminated process and stops as soon as she sees a pair of non-compatible processes sharing a key.

Fix the randomness of the network such that a mismatch occurs. Let P_b be the first (totally honest) process that shares a key with an earlier non-compatible (totally honest) process; let P_a be the latest non-compatible process to terminate that P_b shares a key with[7]. Now if P_b *does not* share a key with any other process, we can easily break π: simulate $\mathcal{M}_M$, opening every process but challenge P_b and answer the challenge by comparing $k*$ with the session key output by P_a. The fact that we do not know whether a mismatch will occur (on this run) and, if so, which processes are P_a and P_b is not a problem: we simply choose two processes uniformly at random at the start of the simulation[8] and treat them as if they will be incompatible and share a session key. We guess the challenge bit correctly either {em whenever }k^* is a random key (regardless if a mismatch occurs or whether the process indices were chosen correctly), or when k^* is the real key, $\mathcal{M}_M$ engineers a mismatch, and we've guessed P_a and P_b correctly. The first event (that k^* is random) happens with probability half, whereas the second happens with significant probability (P_a and P_b are two of at most polynomially many processes, so our random draw will be correct with significant probability).

Of course this does not work when P_b partners with an earlier process, say P_m: simulating $\mathcal{M}_M$ requires that we open P_m, but we can not do so and also challenge P_b (as we would be challenging a partner of an opened process and our guess would be forced to the output of a random coin toss). Note that P_m also partners with P_a (or else P_m and P_a would have been an earlier pair of (totally honest) non-compatible processes sharing key, contradicting the choice of P_b). This means that P_a and P_b are of the same type (e.g. $\langle A, B, 0\rangle$) and P_m is of compatible type (e.g. $\langle B, A, 1\rangle$).

If P_a terminates before P_m, we can still easily break π: we do not need to open P_m to learn its key as we already learned the key by opening P_a; this would allow us to challenge P_b and guess based on if k^* matches P_a's session key. Again, while we do not know which processes are P_a,P_m and P_b, drawing three processes uniformly at random and treating these processes as correct works for the same reason as in the preceding case.

The most difficult and subtle case is when P_m terminates before both P_a and P_b. We cannot open P_m if we want to challenge either P_a or P_b. Intuitively, if $\mathcal{M}_M$ *needs* to see the key output by P_m in order to arrange for both P_a and P_b to output the same key, then we can use this fact to distinguish P_m's output from a random key.

Specifically, we consider a sequence of hybrid experiments where, at each stage, exchanged keys are replaced with genuinely random keys (while preserving partnering). If there is a significant difference in the probability of a triple occurring between hybrids, we have a means of distinguishing between real and random keys. On the other hand, if $\mathcal{M}_M$ can still engineer a triple involving P_m when given a random key in place of P_m's key, then we can simulate $\mathcal{M}_M$ to completion without opening P_m, which enables us to open P_a and challenge P_b.

Now let us be precise. Let $M^1_\pi(\mathcal{M})$ be the event that $\mathcal{M}$ engineers a triple where P_b does not partner with any earlier process, $M^2_\pi(\mathcal{M})$ be the event that P_a and P_b partner with P_m, but that P_m terminates after P_a, and $M^3_\pi(\mathcal{M})$ the event that P_a and P_b partner with P_m and that P_m is the first of these processes to terminate.

[7] We only need the condition "latest" to avoid ambiguity in the (easy) case that processes $\langle A, B, 0\rangle$ and $\langle B, A, 1\rangle$ share a key which is later also shared with, say, a $\langle C, D, 0\rangle$ process

[8] the polynomial upperbound on $\mathcal{M}_M$'s runtime gives us a polynomial upperbound on the maximum number of processes in the network

Lemma B.1. *π is penalty-secure $\implies$ $\forall \mathcal{M}$ $\Pr[M_\pi^1(\mathcal{M})]$ is negligible*

Lemma B.2. *π is penalty-secure $\implies$ $\forall \mathcal{M}$ $\Pr[M_\pi^2(\mathcal{M})]$ is negligible*

Lemma B.3. *π is penalty-secure $\implies$ $\forall \mathcal{M}$ $\Pr[M_\pi^3(\mathcal{M})]$ is negligible*

Lemmas B.1 and B.2 are straightforward and their proofs are omitted. We give a complete proof of Lemma B.3.

Proof. Consider the following modified network ring-master, $\mathcal{RM}_i$. The only modification of $\mathcal{RM}_i$ is how it computes the key it returns in response to an `open` query. $\mathcal{RM}^i$ begins by setting a counter, counter to zero. When a process P terminates, $\mathcal{RM}^i$ calculates it's key as follows:

- if it partners with an earlier process, $\mathcal{RM}_i$ responds with whatever key was output by that process;
- otherwise
  - if counter $< i$, $\mathcal{RM}_i$ draws $k \in_R \{0,1\}^n$, stores this alongside P and reports k as P's keys
  - otherwise, $\mathcal{RM}_i$ returns whatever key P actually output

Now suppose $\mathcal{M}_M$ is an adversary for which $\Pr[M_\pi^3(\mathcal{M}_M)]$ is significant. Let E_i denote the event that $\mathcal{M}_M$ engineers a mismatch of type 3 to occur while interacting with $\mathcal{RM}^i$. Note that E_0 is simply $M_\pi^3(\mathcal{M}_M)$. Let $p(n)$ be a polynomial upper-bounding $\mathcal{M}_M$'s runtime; note that $p(n)$ upper bounds the number of processes started by $\mathcal{M}_M$. Consider the following value:

Case* B.3.1*: $|\Pr[E_0] - \Pr[E_{p(n)}]|$ is significant

Since we have a polynomially long sequence of hybrids, there exists an i such that

$$\Pr[E_i] - \Pr[E_{i+1}] \text{ is significant} \tag{2}$$

We will assume that $\Pr[E_i] - \Pr[E_{i+1}] > 0$ (if this is not the case, invert the bit output by the following adversary).

Let A_i' be the adversary that behaves as follows: Draw $j \in_R [p(n)]$, set counter $= 0$; we simulate $\mathcal{M}_M$ by forwarding all queries but $\texttt{open}(A, B, b, \ell)$ to $\mathcal{RM}$; simulate $\texttt{open}(A, B, b, l)$ by transforming some of the keys. $\mathcal{M}_i'$ we define a key transform map $T : \{0,1\}^n \to \{0,1\}^n$ throughout the simulation as follows:

- if counter $< j$,
  1. issue $\texttt{open}(A, B, b, l)$ to $\mathcal{RM}$, receiving k
  2. if $T(k)$ is defined, return this value
  3. otherwise, increment counter, draw $k' \in_R \{0,1\}^n$, set $T(k) = k'$, and return k'.
- if counter $= j$
  - if strictly less or strictly more than i distinct keys have been returned in the j `open` queries issued so far, abort by outputting a random bit
  - $\texttt{challenge}(A, B, b, l)$, returning k^* as $\langle A, B, b\rangle_l$'s keys
- if counter $> j$,

1. forward $\texttt{open}(A, B, b, l)$ to $\mathcal{RM}$ receiving k
2. if T is defined on k, return $T(k)$; otherwise return k unmodified.

When $\mathcal{M}$ terminates, $\mathcal{M}'_i$ guesses the challenge bit based on whether a mismatch of type 3 appeared; if such a mismatch occurred, guess that k^* was the real key, otherwise guess that k^* was a random key.

If should be clear that whenever $\mathcal{M}'_i$ simulates the interaction of $\mathcal{M}_M$ with $\mathcal{RM}_i$ perfectly, $\mathcal{M}'_i$ wins with $1/2$ plus the quantity of (2), which is significantly more than $1/2$.

$\mathcal{M}'_i$'s simulation is perfect when the jth process to terminate just so happens to be the ith process to output a distinct key, and deviating otherwise. If strictly less or strictly more than i distinct keys have been returned, $\mathcal{M}'_i$ knows this and aborts; the question is what happens if process j's key is not different from the i keys output so far. Note that if $\Pr[M^1_\pi(\mathcal{M}'_i)]$ (resp. $\Pr[M^2_\pi(\mathcal{M}'_i)]$) is significant, then π is insecure by Lemma B.1 (resp. Lemma B.2). Thus we can assume that if the jth process outputs a key that matches one of the earlier i distinct keys, it *partners* with an earlier process and $\mathcal{M}'_i$ is penalized for challenging the jth process, its output is forced to be a random coin-flip, and we are in the same situation as if $\mathcal{M}'_i$ had explicitly aborted.

Of course in any play of $\mathcal{M}_M$ in the full-information game, there is a unique process that is the first to output the ith distinct key; since the choice of j is made obliviously of any of $\mathcal{M}_M$'s choices, then j is correct with significant probability. The result follows.

***Case* B.3.2:** $|\Pr[E^0] - \Pr[E^{p(n)}]|$ is negligible

Let $\mathcal{M}'$ be the following adversary: Draw $i_1, i_2, i_3 \in_R [p(n)]$, letting $i_1 < i_2 < i_3$. We will refer to the i_1st (resp. i_2nd, i_3rd) process to terminate as P_1 (resp. P_2, P_3). Fix $k \in_R \{0,1\}^n$ Set counter $= 0$. Simulate $\mathcal{M}$, forwarding all queries but $\texttt{open}(A, B, b, i)$ to the adversary, which we simulate as follows:

- increment counter.
- if counter $= i_3$, halt
- if counter $= \{i_1, i_2\}$, return k

If P_1,P_2,P_3 are not processes of types that would form a mismatch of type 3 (i.e. it is not the case that $(\texttt{id}(P_1), \texttt{id}(P_2), \texttt{id}(P_3)) \neq ((A, B, b, i), (B, A, \overline{b}, j), (B, A, \overline{b}, k))$ for some names A, B, some bit b and some indices $i, j, k \in [p(n)]$, abort by outputting a random bit. If we haven't aborted, complete the penalty-security game as follows: $\texttt{open}\ P_2$ receiving k_2, $\texttt{challenge}P_3$ receiving $k*$, output "real" if $k^* = k_2$ and "random" if otherwise.

Let E be the event that a triple would occur if $\mathcal{M}_M$ is simulated correctly, and the processes P_1, P_2, P_3 were correctly chosen to be the three processes involved in the triple. It is clear to see that when E occurs, $\mathcal{M}'$ wins the penalty security game with probability negligibly far from 1. It is also straightforward to see that when $\overline{E}$ occurs, $\mathcal{M}'$ wins the penalty security game with probability negligibly close to $1/2$: if $\mathcal{M}'$ is not penalized, these processes will have output different keys, and so $\mathcal{M}'$ will output "random" with probability negligibly close to 1. Finally, given that the choices of i_1, i_2, i_3 are drawn uniformly at random and independently of any of $\mathcal{M}$ or $\mathcal{RM}$'s actions, P_1, P_2, P_3 are the correct processes with probability $1/p^3(n)$, and so the probability that E occurs is significant. □

C Proof of Theorem 3.1

Proof of Theorem 3.1(⇐): Suppose $\mathcal{MP}$ breaks the partner-security of π; our goal is to break the penalty-security of π. By Theorem 3.3 we can assume, except with negligible probability, that in all networks arranged by $\mathcal{MP}$ at most two totally honest processes output the same key, and when this occurs, these processes are compatible. In any play of the partner-security game there is a unique process, say P_{CH}, that is challenged by $\mathcal{MP}$. Since $\mathcal{MP}$ can ask whether compatible processes partner, we assume, without loss of generality, that no totally honest process terminating before P_{CH} has output the same session key as P_CH.

Now if we knew in advance which process is P_{CH}, we could simulate $\mathcal{MP}$ as follows: to answer partnering queries, we compute an "effective session key" for each process and report processes as partners if they are compatible and have the same effective key. The effective session key of P_{CH} is computed as soon as P_{CH} terminates, and is the result of challenging P_{CH}; for all other processes, the effective session key is whatever is session key is returned from the ring-master when we ask to open that process. When $\mathcal{MP}$ challenges P_{CH} we give k^* as the challenge, continue simulating $\mathcal{MP}$ until completion and echo $\mathcal{MP}$'s guess. If $\mathcal{MP}$ asks to open a process, we return that process's effective session key.

The correctness of this simulation relies on us reporting partnering correctly. Clearly all partnering queries concerning effective keys different from k^* are answered correctly, and even those concerning k^* are answered correctly when k^* is the real session key (output by P_{CH}). So suppose $b^* = 1$, and k^* is a uniformly random key (different from the one output by P_{CH}). Thanks to Theorem 3.3, we know that at most one other totally honest process shares a key with P_{CH}, and if this process exists, it is a compatible process. When we ask to open this process, by the definition of the penalty-security game, we are given k^*, so we correctly report this process as the partner of P_{CH}. Further, Lemma 3.2 guarantees that no totally honest process actually outputs k^*, and so the only time we see k^* is if it has been substituted as the process's key by the ring-master.

It remains to show that we can effectively do this simulation without knowing (a priori) which process is P_{CH}. We will choose a process uniformly at random at the start of the simulation, say P_R, and treat this process as if it is P_{CH}. Now when P_R happens to be P_{CH}, we simulate $\mathcal{MP}$ to completion, breaking the penalty-security of π with the same success probability of $\mathcal{MP}$ attacking the partner-security. Probably P_R will not be this desired process.

It is bad if P_R partners with a process that terminated earlier (recall that, as an adversary playing the penalty game, we have no way of knowing this fact). If, when we challenge P_R, we get a random k^*, we will subsequently incorrectly answer some partnering queries (specifically, if $\mathcal{MP}$ asks if P_R partners with an earlier process, we would incorrectly answer no). However, according to the rules of the penalty-security game, by challenging P_R, which partners with an earlier process, we are forced to stop the security experiment flip a coin as our guess, stopping us from otherwise giving a wrong simulation of $\mathcal{MP}$.

So suppose otherwise (i.e. that P_R does not partner with an earlier process). If $\mathcal{MP}$ challenges a process other than P_R we 'abort' our simulation by outputting a random bit. If, before $\mathcal{MP}$ challenges a process, $\mathcal{MP}$ asks to open a process compatible with P_R and whose effective session key is k^*, then again we abort, because we know that $\mathcal{MP}$ could not later challenged P_R.

Since P_R is chosen from a set of polynomial size, P_R will be the right process with significant probability; in all other cases, we win with probability exactly 1/2. Taken together, these facts imply that that π is not penalty-secure. □