Constructions of Truly Practical Secure Protocols using Standard Smartcards[*]

Carmit Hazay[†] Yehuda Lindell[†]

January 20, 2009

Abstract

In this paper we show that using standard smartcards it is possible to construct truly practical secure protocols for a variety of tasks. Our protocols achieve full *simulation-based security* in the presence of *malicious adversaries*, and can be run on very large inputs. We present protocols for secure set intersection, oblivious database search and more. We have also implemented our set intersection protocol in order to show that it is truly practical: on sets of size 30,000 elements takes 20 seconds for one party and 30 minutes for the other (where the latter can be parallelized to further reduce the time). This demonstrates that in settings where physical smartcards can be sent between parties (as in the case of private data mining tasks between security and governmental agencies), it is possible to use secure protocols with proven simulation-based security.

1 Introduction

In the setting of secure multiparty computation, a set of parties with private inputs wish to jointly compute some functionality of their inputs. Loosely speaking, the security requirements of such a computation are that (i) nothing is learned from the protocol other than the output (privacy), (ii) the output is distributed according to the prescribed functionality (correctness), and (iii) parties cannot make their inputs depend on other parties' inputs. The standard definition of security is based on comparing a real protocol execution to an ideal execution where a trusted party carries out the computation for the parties. This notion is typically called *simulation-based security.* Secure multiparty computation forms the basis for a multitude of tasks, including those as simple as coin-tossing and agreement, and as complex as electronic voting and auctions, electronic cash schemes, anonymous transactions, remote game playing (a.k.a. "mental poker"), and privacy-preserving data mining.

The security requirements in the setting of multiparty computation must hold even when some of the participating parties are adversarial. In this paper, we consider malicious adversaries that can arbitrarily deviate from the protocol specification. It has been shown that, with the aid of suitable cryptographic tools, *any* two-party or multiparty function can be securely computed [24, 14, 13, 4, 7] in the presence of malicious adversaries. However, protocols that achieve this level of security are rarely efficient enough to be used in practice, even for relatively small inputs.

[*]An extended abstract of this work appeared in the 15*th ACM Conference on Computer and Communications Security (ACM CCS),* 2008.

[†]Department of Computer Science, Bar-Ilan University, Israel. Email: {harelc,lindell}@cs.biu.ac.il. This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 781/07). The first author is also supported by an Eshkol scholarship from the Israel Ministry of Science.

Recently, there has been much interest in the data mining and other communities for secure protocols for a wide variety of tasks. This interest exists not only in academic circles, but also in industry, in part due to the growing conflict between the privacy concerns of citizens and the homeland security needs of governments. Unfortunately, however, truly practical protocols that also achieve proven simulation-based security are currently far out of reach. This is especially the case when security in the presence of malicious adversaries is considered (see related work for other models).

Smartcard-aided secure computation. In this paper, we construct protocols that use smartcards in addition to standard network communication. Specifically, in addition to sending messages over a network, the participating parties may initialize smartcards in some way and send them to each other. Of course, such a *modus operandi* is only reasonable if this is not over-used. In all of our protocols, one party initializes a smartcard and sends it to the other, and that is all. Importantly, it is also sufficient to send a smartcard once, which can then be used for many executions of the protocol (and even for different protocols). This model is clearly not suitable for protocols that must be run by ad hoc participants over the Internet (e.g., for secure eBay auctions or secure Internet purchases). However, we argue that it is suitable whenever parties with non-transient relationships need to run secure protocols. Thus, this model is suitable for the purpose of *privacy-preserving data mining* between commercial, governmental and security agencies. We construct practical *two-party* protocols for the following tasks:

- *Secure set intersection:* This problem is of great interest in practice and has many applications. Some examples are: finding out if someone is on two security agencies' list of suspects, finding out if someone illegally receives social welfare from two different agencies, finding out what patients receive medical care at two different medical centers, and so on. This problem has received a lot of attention due to its importance; see [21, 11, 17] for some examples. We present a protocol that is far more efficient than any known current solutions, and provides the highest level of security (full-simulation in the presence of malicious adversaries, and even universal composability). Our protocol is surprisingly simple, and essentially requires one party to carry out one 3DES or AES computation on each set element (using a regular PC), while the other party carries out the same computations using a smartcard. Thus, for sets comprised of 30,000 elements, the first party's computation takes approximately 20 seconds and the second party's computation takes approximately 30 minutes (but can be parallelized, meaning that using 10 smartcards, the computation would take approximately 3 minutes). In our protocol, only the second party receives output.
- *Oblivious database search:* In this problem, a client is able to search a database held by a server so that: (a) the client can only carry out a single search (or a predetermined number of searches authorized by the server), and learns nothing beyond the result of the authorized searches; and (b) the server learns nothing about the searches carried out by the client. We remark that searches are as in the standard database setting: the database has a "key attribute" and each record has a unique key value; searches are then carried out by inputting a key value – if the key exists in the database then the client receives back the entire record; otherwise it receives back a "`non-existent`" reply. This problem has been studied in [8, 10] and has important applications to privacy. For example, consider the case of homeland security where it is sometimes necessary for one organization to search the database of another. In order to minimize information flow (or stated differently, in order to preserve the "need to know" principle), we would like the agency carrying out the search to have access only to the single piece of information it is searching for. Furthermore, we would like the value being searched for

to remain secret. Another, possibly more convincing, application comes from the commercial world. The LexisNexis database is a paid service provided to legal professionals that enables them – among other things – to search legal research and public records, for the purpose of case preparation. Now, the content of searches made for case preparation is *highly confidential*; this information reveals much about the legal strategy of the lawyers preparing the case, and would allow the other side to prepare counter-arguments well ahead of time. It is even possible that revealing the content of some of these searches may breach attorney-client privilege. We conclude that the searches made to LexisNexis must remain confidential, and even LexisNexis should not learn them (either because they may be corrupted, or more likely, a breach to their system could be used to steal this confidential information). Oblivious database search can be used to solve this exact problem. We present a protocol for oblivious database search that reaches a level of efficiency that is almost equivalent to a non-private database search. Once again, we achieve provable security (under full simulation-based definitions) in the presence of malicious adversaries.

- *Oblivious document search:* A similar, but seemingly more difficult, problem to that of oblivious database search is that of oblivious document search. Here, the database is made up of a series of unstructured documents and a keyword query should return all documents that contain that query. This is somewhat more difficult than the previous problem because of the dependence between documents (the client should not know if different documents contain the same keyword if it has not searched them both). Nevertheless, using smartcards, we present a highly efficient protocol for this problem, that is provably secure in the presence of malicious adversaries. We remark that in many cases, including the LexisNexis example above, what is really needed is the unstructured document search here.

We stress that our protocols are all proven secure under the standard simulation-based definition of security (cf. [5, 13] following [15, 3, 20]), and for the case of *malicious adversaries* that may follow any arbitrary polynomial-time strategy. Thus, the highest level of security is achieved. As we have mentioned, however, we use a smartcard to aid in the computation, unlike the standard model of computation. As will become clear, this gives extraordinary power and makes it possible to construct protocols that are far more efficient than anything previously known.

Composability. One criticism of attempts to construct secure protocols that are to be used in practice is that the stand-alone model (where security is proven for only a single execution of a protocol in isolation – or equivalently when the adversary is assumed to attack only a single execution) is not the real-world model of computation. Thus, why does it make sense to insist on a full proof of security when the proof is for an unrealistic model? Fortunately, all of our protocols are secure under concurrent general composition (or equivalently, universal composability), and thus their proven security is guaranteed in the real-world setting that they may be used.

Standard smartcards – what and why. We stress that our protocols are designed so that any standard smartcard can be used. Before proceeding we explain why it is important for us to use *standard* – rather than special-purpose – smartcards, and what functionality is provided by such standard smartcards. The reason for our insistence on standard smartcards is twofold:

1. *Ease of deployment:* It is much easier to actually deploy a protocol that uses standard smartcard technology. This is due to the fact that many organizations have already deployed smartcards, typically for authenticating users. However, even if this is not the case, it is

possible to purchase any smartcard from essentially any smartcard vendor.[1]

2. *Trust:* If a special-purpose smartcard needs to be used for a secure protocol, then we need to trust the vendor who built the smartcard. This trust extends to believing that they did not incorrectly implement the smartcard functionality on purpose or unintentionally. In contrast, if standard smartcards can be used then it is possible to use smartcards constructed by a third-party vendor (and possibly constructed before our protocols were even designed). In addition to reducing the chance of malicious implementation, the chance of an unintentional error is much smaller, because these cards have been tried and tested over many years.

We remark that Javacards can also be considered for the application that we are considering. Javacards are smartcards with the property that special-purpose Java applets can be loaded onto them in order to provide special-purpose functionality. We remark that such solutions are also reasonable. However, it does make deployment slightly more difficult as already-deployed smartcards (that are used for smartcard logon and VPN authentication for example) cannot be used. Furthermore, it is necessary to completely trust whoever wrote the applet; this can be remedied by having an open-source applet which can be checked before loaded. Therefore, protocols that do need smartcards with some special-purpose functionality can be used, but are slightly less desirable.

A trusted party? At first sight, it may seem that we have essentially introduced a trusted party into the model, and so of course everything becomes easy. We argue that this is not the case. First, a smartcard is a very specific type of trusted party, with very specific functionality (especially if we focus on standard smartcards). Second, due to it being weak hardware, a smartcard cannot carry out a computation on large inputs. Thus, even a special-purpose smartcard cannot directly compute set intersection on inputs of size 30,000. Finally, smartcards are used in practice and are becoming more and more ubiquitous. Thus, our model truly is a realistic one, and our protocols can easily be deployed in practice.

Trusting smartcards. In our protocols, we assume that the smartcard is uncorrupted. We base this assumption on the fact that *modern* smartcards are widely deployed today – mostly for authentication – and are rarely broken (we stress that we refer to smartcards that have passed certification like FIPS or Common Criteria, and not microprocessors with basic protection). We discuss the security of smartcards in more detail at the end of Section 2.

Smartcard authenticity. As we have mentioned, our protocols require one party to initialize a smartcard and send it to the other. Furthermore, the recipient of the smartcard needs to trust that the device that it receives is really a smartcard of the specified type. Since our protocols rely on standard smartcard technology only, this problem essentially reduces to identifying that a given device was manufactured by a specified smartcard vendor. In principle, this problem is easily solved by having smartcard manufacturers initialize all devices with a public/private key pair, where the private key is known only to the manufacturer. Then, given a device and the manufacturer's public key it is possible to verify that the device is authentic using a simple challenge/response mechanism. This solution is not perfect because given the compromise of a single smartcard, it is possible to manufacture multiple forged devices. This is highly undesirable because it means that the incentive to carry out such an attack can be very high. This can be improved by using different public keys for

[1]Of course, the notion of a "standard" smartcard is somewhat problematic because different vendors construct smartcards with different properties. We therefore rely on properties that we know are in the widely-used smartcards sold by Siemens.

different batches (or even a different key for every device, although this is probably too cumbersome in practice). To the best of our knowledge, such a mechanism is typically not implemented today (rather, symmetric keys are used instead). Nevertheless, it could be implemented without much difficulty and so is not a serious barrier.

Related work. Secure computation has been studied at great length for over two decades. However, the study of highly-efficient protocols for problems of interest has recently been intensively studied under the premise of "privacy-preserving data mining", starting with [19]. Most of the secure protocols for this setting have considered the setting of semi-honest adversarial behavior, which is often not sufficient. Indeed, highly-efficient protocols that are proven secure in the presence of malicious adversaries and using the simulation-based approach are few and far between; one notable exception being the work of [1] for securely computing the median. Therefore, researchers have considered other directions. One possibility is to consider privacy only; see for example [9, 21, 6]. A different direction considered recently has been to look at an alternative adversary model that guarantees that if an adversary cheats then it will be caught with some probability [2, 16]. We stress that our protocols are more efficient than all of the above and also reach a higher level of security than most. (Of course, we have the additional requirement of a smartcard and thus a comparison of our protocols is not really in place; rather we view this as a comparison of models.)

2 Standard Smartcard Functionality and Security

In this section we describe what functionality is provided by standard smartcards, and the security guarantees provided by them. Our description of standard smartcard functionality does not include an exhaustive list of all available functions. Rather we describe the most basic functionality and some additional specific properties that we use:

1. *On-board cryptographic operations:* Smartcards can store cryptographic keys for private and public-key operations. Private keys that are stored (for decryption or signing/MACing) can only be used according to their specified operation and *cannot* be exported. We note that symmetric keys are always generated outside of the smartcard and then imported, whereas asymmetric keys can either be imported or generated on-board (in which case, no one can ever know the private key). Two important operations that smartcards can carry out are basic block cipher operations and CBC-MAC computation. These operations may be viewed as pseudorandom function computations, and we will use them as such. The symmetric algorithms typically supported by smartcards use 3DES and/or AES, and the asymmetric algorithms use RSA (with some also supporting Elliptic curve operations).
2. *Authenticated operations:* It is possible to "protect" a cryptographic operation by a logical test. In order to pass such a test, the user must either present a password or pass a challenge/response test (in the latter case, the smartcard outputs a random challenge and the user must reply with a response based on some cryptographic operation using a password or key applied to the random challenge).
3. *Access conditions:* It is possible to define what operations on a key are allowed and what are not allowed. There is great granularity here. For all operations (e.g., use key, delete key, change key and so on), it is possible to define that no one is ever allowed, anyone is allowed, or only a party passing some test is allowed. We stress that for different operations (like use and delete) a different test (e.g., a different password) can also be defined.

4. *Special access conditions:* There are a number of special operations; we mention two here. The first is a *usage counter*; such a counter is defined when a key is either generated or imported and it says how many times the key can be used before it “expires”. Once the key has expired it can only be deleted. The second is an *access-granted counter* and is the same as a usage counter except that it defines how many times a key can be used after passing a test, before the test must be passed again. For example, setting the access-granted counter to 1 means that the test (e.g., passing a challenge/response) must be passed every time the key is used.
5. *Secure messaging:* Operations can be protected by “secure messaging” which means that all data is encrypted and/or authenticated by a private (symmetric) key that was previously imported to the smartcard. An important property of secure messaging is that it is possible to receive a “receipt” testifying to the fact that the operation was carried out; when secure messaging with message authentication is used, this receipt cannot be tampered with by a man-in-the-middle adversary. Thus, it is possible for one party to initialize a smartcard and send it to another party, with the property that the first party can still carry out secure operations with the smartcard without the second party being able to learn anything or tamper with the communication in an undetected way. One example where this may be useful is that the first party can import a secret key to the smartcard without the second party who physically holds the card learning the key. We remark that it is typically possible to define a different key for secure messaging that is applied to messages being sent *to* the smartcard and to messages that are received *from* the smartcard (and thus it is possible to have unidirectional secure messaging only). In addition to privacy, secure messaging can be used to ensure *integrity.* Thus, a message authentication code (MAC) can be used on commands to the smartcard and responses from the smartcard. This can be used, for example, to enables a remote user to verify that a command was issued to the smartcard by the party physically holding the smartcard. (In order to implement this, a MAC is applied to the smartcard-response to the command and this MAC is forwarded to the remote user. Since it is not possible to forge a MAC without knowing the secret key, the party physically holding the smartcard cannot forge a response and so must issue the command, as required.)
6. *Store files:* A smartcard can also be used to store files. Such files can either be public (meaning anyone can read them) or private (meaning that some test must be passed in order to read the file). We stress that private keys are not files because such a key can never be read out of a smartcard. In contrast a public key is essentially a file.

We stress that all reasonable smartcards have all of the above properties, with the possible exception of the special access conditions mentioned above in item 4. We do not have personal knowledge of any smartcard that does not, but are not familiar with all smartcard vendors. We do know that the smartcards of Siemens (and others) have these two counters.

Smartcard Security. We conclude this section by remarking that smartcards provide a high level of *physical security.* They are not just a regular microcontroller with defined functionality. Rather, great progress has been made over the years to make it very hard to access the internal memory of a smartcard. Typical countermeasures against physical attacks on a smartcard include: shrinking the size of transistors and wires to 200nm (making them too small for analysis by optical microscopes and too small for probes to be placed on the wires), multiple layering (enabling sensitive areas to be buried beneath other layers of the controller), protective layering (a grid is placed around the smartcard and if this is cut, then the chip automatically erases all of its memory), sensors (if the light, temperature etc. are not as expected then again all internal memory is immediately destroyed), bus scrambling (obfuscating the communication over the data bus between different components to make it hard to interpret without full reverse engineering), and glue logic (mixing up components of the controller in random ways to make it hard to know what components hold what functionality). For more information, we refer the reader to [23]. Having said the above, there is no perfect security mechanism and this includes smartcards. Nevertheless, we strongly believe that it is a reasonable assumption to trust the security of high-end smartcards (for example, smartcards that have FIPS 140-2, level 3 or 4 certification). Our belief is also supported by the computer-security industry: smartcards are widely used today as an authentication mechanism to protect security-critical applications.

3 Definitions and Tools

We use the standard definition of two-party computation for the case of no honest majority, where no fairness is guaranteed. In particular, this means that the adversary always receives output first, and can then decide if the honest party also receives output; this is called "security with abort" because a corrupted party can abort after receiving output and prevent the honest party from also receiving output. We refer the reader to [13, Section 7] for full definitions of security for secure two-party computation, and present a very brief description here only.

Preliminaries. A function $\mu(\cdot)$ is negligible in n, or just negligible, if for every positive polynomial $p(\cdot)$ and all sufficiently large n's, $\mu(n) < 1/p(n)$. A probability ensemble $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathsf{N}}$ is an infinite sequence of random variables indexed by a and $n \in \mathsf{N}$. (The value a will represent the parties' inputs and n the security parameter.) Two distribution ensembles $X = \{X(a,n)\}_{n\in\mathsf{N}}$ and $Y = \{Y(a,n)\}_{n\in\mathsf{N}}$ are said to be computationally indistinguishable, denoted $X \overset{c}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$,

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \leq \mu(n)$$

All parties run in time that is polynomial in the security parameter. (Formally, each party has a security parameter tape upon which the value 1^n is written. Then the party is polynomial in the input on this tape.)

Communication model. In this paper we consider a setting where parties can interact with each other and with a physical smartcard. We model these interactions in the usual way. Specifically, each party has two outgoing communication tapes and two incoming communication tapes; one for interacting with the other party and one for interacting with a smartcard. Of course, only the party physically holding the smartcard can interact with it via its communication tapes (if the other party wishes to send a message to the smartcard it can only do so by sending it via the party holding the smartcard). This model accurately reflects the real-world scenario of interactive computation with smartcards.

Secure two-party computation. A two-party protocol problem is cast by specifying a random process that maps sets of inputs to sets of outputs (one for each party). This process is called a functionality and is denoted $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, where party P_1 is supposed to receive the first output and party P_2 the second output. We consider the case of malicious adversaries (who may arbitrarily deviate from the protocol specification) and static corruptions (meaning that the party controlled by the adversary is fixed before the execution begins).

Security is formalized by comparing a real protocol execution to an ideal model setting where a trusted party is used to carry out the computation. In this ideal model, the parties send their inputs to the trusted party who first sends the output to the adversary. (The adversary controls one of the parties and can instruct it to behave arbitrarily). After the adversary receives the output it either sends continue to the trusted party instructing it to also send the output to the honest party, or halt in which case the trusted party sends $\perp$ to the honest party. The honest party outputs whatever it received from the trusted party and the adversary outputs whatever it wishes. We stress that the communication between the parties and the trusted party is ideally secure. The pair of outputs of the honest party and an adversary $\mathcal{A}$ in an ideal execution where the trusted party computes f is denoted $\text{IDEAL}_{f,\mathcal{A}(z)}(x_1, x_2, n)$, where x_1, x_2 are the respective inputs of P_1 and P_2, z is an auxiliary input received by $\mathcal{A}$ (representing any prior knowledge $\mathcal{A}$ may have about the honest party's input), and n is the security parameter.

In contrast, in the real model, a real protocol π is run between the parties without any trusted help. Once again, an adversary $\mathcal{A}$ controls one of the parties and can instruct it to behave arbitrarily. At the end of the execution, the honest party outputs the output specified by the protocol π and the adversary outputs whatever it wishes. The pair of outputs of the honest party and an adversary $\mathcal{A}$ in an real execution of a protocol π is denoted $\text{REAL}_{\pi,\mathcal{A}(z)}(x_1, x_2, n)$, where x_1, x_2, z and n are as above.

Given the above, we can now define the security of a protocol π.

Definition 1 *Let f and π be as above. Protocol π is said to* securely compute f with abort in the presence of malicious adversaries *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model, such that for every $I \subseteq [2]$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),I}(x, y, n)\right\}_{x,y,z\in\{0,1\}^*,n\in\mathsf{N}} \overset{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(x, y, n)\right\}_{x,y,z\in\{0,1\}^*,n\in\mathsf{N}}$$

where $|x| = |y|$.

Reactive functionalities. In some cases, the computation carried out by the trusted party is not a simple function mapping a pair of inputs to a pair of outputs. Rather, it can be a more complex computation that consists of a number of phases where inputs are received and outputs are sent (e.g., think of secure poker; parties receive cards, chooses which cards to throw, and then receive more cards). Such a computation is called a reactive functionality.

Message authentication codes. Informally speaking, a message authentication code (MAC) is the symmetric analogue of digital signatures. Specifically, given the shared secret key it is possible to generate a MAC tag whose legitimacy can be verified by anyone else knowing the secret key. A MAC is said to be secure if without knowledge of the key, no polynomial-time adversary can generate a tag that will be accepted, except with negligible probability; see [13] for a formal definition.

Pseudorandom permutations and smartcards. Informally speaking, a pseudorandom permutation is an efficiently computable bijective function that looks like a truly random bijective function to any polynomial-time observer; see [12] for a formal definition. We remark that pseudorandom permutations have short secret keys and they look like random functions to any observer that does not know the key. Modern block ciphers like 3DES and AES are assumed to be pseudorandom permutations (and indeed one of the criteria in the choice of AES was that it should be indistinguishable from a random permutation).

One of the basic cryptographic operations of any smartcard is the computation of a block cipher using a secret key that was imported into the smartcard (and is never exported from it later). We use pseudorandom permutations in our protocols and will assume that the block cipher in the smartcard behaves like a pseudorandom permutation. This is widely accepted for modern block ciphers, and in particular for 3DES and AES. We remark that this assumes that the size of inputs to the pseudorandom permutation are of the appropriate size (e.g., 128 bits for AES).

4 Secure Set Intersection

In this section we show how to securely compute the secure set intersection problem defined by $F_\cap(X,Y) = X \cap Y$, where $X = \{x_1,\ldots,x_{n_1}\}$ and $Y = \{y_1,\ldots,y_{n_2}\}$, and one party receives output (while the other learns nothing). We note that the problem of securely computing the function f_{eq}, defined as $f_{\text{eq}}(x,y) = 1$ if and only if $x = y$, is a special case of set intersection. Thus, our protocol can also be used to compute f_{eq} with extremely high efficiency.

The basic idea behind our protocol is as follows. The first party P_1, with input set $X = \{x_1,\ldots,x_{n_1}\}$ initializes a smartcard with a secret key k for a pseudorandom permutation F (i.e., F is a block cipher). Then, it computes $X_F = \{F_k(x_1),\ldots,F_k(x_{n_1})\}$ and sends X_F and the smartcard to the second party. The second party P_2, with input $Y = \{y_1,\ldots,y_{n_2}\}$ then uses the smartcard to compute $F_k(y_i)$ for every i, and it outputs every y_i for which $F_k(y_i) \in X_F$. It is clear that P_1 learns nothing because it does not receive anything in the protocol. Regarding P_2, if it uses the smartcard to compute $F_k(y)$ for some $y \in X \cap Y$, then it learns that $y \in X$, but this is the information that is supposed to be revealed! In contrast, for every $x \in X$ that for which P_2 does not use the smartcard to compute $F_k(x)$, it learns nothing about x from X_F (because $F_k(x)$ just looks like a random value).

Despite the above intuitive security argument, there are a number of subtleties that arise. First, nothing can stop P_2 from asking the smartcard to compute $F_k(y)$ for a huge number of y's (taking this to an extreme, if X and Y are social security numbers, then P_2 can use the smartcard to compute the permutation on all possible social security numbers). We prevent this by having P_1 initialize the key k on the smartcard with a usage counter set to n_2. Recall that this means that the key k can be used at most n_2 times, after which the key can only be deleted. In addition to the above, in order to achieve simulation-based security we need to have party P_2 compute $F_k(y)$ for all $y \in Y$ *before* P_1 sends it X_F (this is a technicality that comes out of the proof). In order to achieve this, we have P_1 initialize k with secure messaging for authentication using an additional key k_{init}. This initialization is an association between the key k and the key k_{init} so that when a command to delete k is issued to the smartcard, the confirmation by the smartcard that this operation took place is authenticated using a message authentication code keyed with k_{init} (standard smartcards support such a configuration). Observe that given this initialization, P_2 can prove to P_1 that it has deleted k before P_1 sends X_F (note that P_1 knows k_{init} and so can verify that the MAC is correct).

4.1 The Basic Protocol

Let F be a pseudorandom permutation with domain $\{0,1\}^n$ and keys that are chosen uniformly from $\{0,1\}^n$ (this is for simplicity only).

Protocol 2 (secure set intersection – P_2 only receives output)

- **Inputs:** *Party P_1 has a set of n_1 elements and party P_2 has a set of n_2 elements; all elements are taken from $\{0,1\}^n$, where n also serves as the security parameter.*
- **Auxiliary inputs:** *Both P_1 and P_2 are given n_1 and n_2, as well as the security parameter n.*
- **SmartCard Initialization:** *Party P_1 chooses two keys $k, k_{\text{init}} \leftarrow \{0,1\}^n$ and imports k into a smartcard SC for usage as a pseudorandom permutation. P_2 sets the usage counter of k to be n_2 and defines that the confirmation to* `DeleteObject` *is MACed using the key k_{init}.*
  P_1 sends SC to P_2 (this takes place before the protocol below begins).[2]
- **The protocol:**
  1. P_2's first step:
     (a) *Given the smartcard SC, party P_2 computes the set $Y_F = \{(y, F_k(y))\}_{y \in Y}$.*
     (b) *Next, P_2 issues a* `DeleteObject` *command to the smartcard to delete k and receives back the confirmation from the smartcard.*
     (c) *P_2 sends the delete confirmation to P_1.*
  2. P_1's step: *P_1 checks that the* `DeleteObject` *confirmation states that the operation was successful and verifies the MAC-tag on the response. If either of these checks fail, then P_1 outputs $\perp$ and halts. Otherwise, it computes the set $X_F = \{F_k(x)\}_{x \in X}$, sends it to P_2 and halts.*
  3. P_2's second step: *P_2 outputs the set $\{y \mid F_k(y) \in X_F\}$ and halts.*

We have the following theorem:

Theorem 3 *Assume that F is a pseudorandom permutation over $\{0,1\}^n$. Then, Protocol 2 securely computes the function $F_\cap(X, Y) = X \cap Y$ in the presence of malicious adversaries, where only P_2 receives output.*

Proof: We treat each corruption case separately:

No parties are corrupted. In this case, all the adversary sees is the list X_F which reveals nothing about X by the fact that F is a pseudorandom permutation (recall that we assume that the adversary cannot intercept and use the smartcard while en route between the parties).

Party P_1 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_1. We construct an ideal-model simulator $\mathcal{S}$ that works with a trusted party computing $F_\cap$. $\mathcal{S}$ invokes $\mathcal{A}$ upon its input and receives from $\mathcal{A}$ the keys k and k_{init} that $\mathcal{S}$ imports to the smartcard ($\mathcal{S}$ receives all messages sent by $\mathcal{A}$, including those sent to the smartcard, because sending a message to the smartcard involves $\mathcal{A}$ writing on its outgoing communication tape to the smartcard which can be read by $\mathcal{S}$). Then, $\mathcal{S}$ hands $\mathcal{A}$ a confirmation message for delete with a correct MAC (computed using k_{init}). Following this, $\mathcal{S}$ receives $X_F = \{z_1, \ldots, z_{n_1}\}$ from $\mathcal{A}$ and for every i sets $x_i = F_k^{-1}(z_i)$. (If some z_i is not in the range of F_k, then $\mathcal{S}$ ignores it.) Finally, $\mathcal{S}$ sends $X = \{x_1, \ldots, x_{n_1}\}$ to the trusted party computing $F_\cap$, outputs whatever $\mathcal{A}$ outputs and halts. The view of $\mathcal{A}$ and thus its output in this simulation is identical to a real execution because it consists only of the delete confirmation

[2] We assume that SC is sent via a secure carrier and so cannot be accessed by an adversary in the case that P_1 and P_2 are both honest. This assumption can be removed by protecting the use of k with a random password of length n. Then, P_1 sends the password to P_2 after it receives SC.

message. Regarding the output of the honest P_2, notice that in a real execution P_2 outputs an element y if and only if $F_k(y) \in X_F$ which is equivalent to saying that there exists a $z \in X_F$ such that $F_k^{-1}(z) \in Y$. However, this is exactly what determines P_2's output in the ideal model, as required.

Party P_2 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_2. We construct an ideal-model simulator $\mathcal{S}$ that works with a trusted party computing $F_\cap$. $\mathcal{S}$ chooses random k, k_{init}, initializes $Y = \phi$, and invokes $\mathcal{A}$ upon its input. Whenever $\mathcal{A}$ sends a value y intended for the smartcard, $\mathcal{S}$ adds y to the set Y, and gives $\mathcal{A}$ the smartcard response $F_k(y)$ computed using the key k that $\mathcal{S}$ chose. If $\mathcal{A}$ attempts to send more than n_2 values to the smartcard, $\mathcal{S}$ replies with a `fail` message (simulating what the smartcard would do if the usage counter reaches zero). Finally, $\mathcal{S}$ receives a delete confirmation message from $\mathcal{A}$. If the message is not valid (when checking the MAC with key k_{init}, then $\mathcal{S}$ sends $\perp$ to the trusted party for input, outputs whatever $\mathcal{A}$ outputs and halts. Else, $\mathcal{S}$ sends the set Y that it constructed above to the trusted party and receives back the set $Z = X \cap Y$. Simulator $\mathcal{S}$ then constructs the set X_F by first adding $F_k(z)$ for every $z \in Z$. Then, $\mathcal{S}$ adds $F_k(z)$ for $n_1 - |Z|$ distinct elements that are also different from every element in Y. Finally, $\mathcal{S}$ hands $\mathcal{A}$ the set X_F (as if it was received from P_1, outputs whatever $\mathcal{A}$ outputs and then halts.

We argue that the output distribution of $\mathcal{S}$ and the honest P_2 in the ideal model is computationally indistinguishable from the output distribution of $\mathcal{A}$ and the honest P_2 in a real protocol execution. In order to prove this, we construct $\mathcal{S}'$ who works exactly like $\mathcal{S}$ except that it uses a truly random permutation instead of F_k. Using a straightforward reduction to the security of the pseudorandom permutation, we have that the output of $\mathcal{S}'$ and P_2 is computationally indistinguishable from the output of $\mathcal{S}$ and P_2. Next, we construct $\mathcal{S}''$ who instead of interacting with a trusted third party is given P_1 real input set X. Then, $\mathcal{S}''$ constructs the set X_F like $\mathcal{S}'$ except that the $n_1 - |Z|$ elements that are added are those in the set $X - X \cap Y$ (but again, using a truly random permutation). Since both $\mathcal{S}'$ and $\mathcal{S}''$ construct X_F by applying a random permutation to n_1 distinct elements, we have that the distributions are identical. Finally, we construct $\mathcal{S}'''$ who works exactly like $\mathcal{S}''$ exact that it uses F_k again, instead of using a random permutation. Once again, the output distribution of $\mathcal{S}''$ and P_2 is indistinguishable from the output distribution of $\mathcal{S}'''$ and P_2, due to the assumption that F_k is a pseudorandom permutation. The proof of this corruption case is concluded by noting that the messages sent by $\mathcal{S}'''$ are exactly the same as those sent by an honest P_1. (Note that $\mathcal{S}'''$ constructs X_F by taking the set $Z = X \cap Y$ and then adding $X - X \cap Y$, but this means that it is constructed from the set X, just like an honest P_1.) ■

Composability. Observe that our simulators above do not rewind $\mathcal{A}$ at all. Thus, as shown in [18], this proves that the protocol is also secure under concurrent general composition (equivalently, it is universally composable). We remark that in [18] this is shown only for protocols that have the additional property of "start synchronization". However, this always holds for two-party protocols.

Reusing the smartcard. Although we argue that it is realistic for parties in non-transient relationships to send smartcards to each other, it is not very practical for them to do this every time they wish to run the protocol. Rather, they should be able to do this only once, and then run the protocol many times. This is achieved in a very straightforward way using *secure messaging*. Specifically, P_1 initializes the smartcard so that a key for a pseudorandom permutation can be imported, while encrypted under a secure messaging key k_{sm}. This means that P_1 can begin the protocol by importing a new key k to the smartcard (with usage counter n_2 for the size of the set in this execution and protected with k_{init} for delete as above). This means that P_1 only needs to send a smartcard once to P_2 and the protocol can be run many times, using standard network communication only.

4.2 Experimental Results

We implemented our protocol for set intersection using the eToken smartcard of Aladdin Knowledge Systems and received the following results:

Size of each set	**Run-time of P_1**	**Run-time of P_2**	**Avg time per element for P_2**
1000	2 sec	52 sec	52 ms.
5000	5 sec	262 sec	52 ms.
10000	8 sec	493 sec	49 ms.
20000	14 sec	1196 sec	60 ms.
30000	21 sec	1982 sec	66 ms.

These results confirm the expected complexity of approximately 50 milliseconds per smartcard operation. We remark that no code optimizations were made and the running-time can be further improved (although the majority of the work is with the smartcard and this cannot be made faster without further improvements in smartcard technology).

5 Oblivious Database Search

In this section we study the problem of oblivious database search. The aim here is to allow a client to search a database without the server learning the query (or queries) made by the client. Furthermore, the client should only be able to make a single query (or, to be more exact, the client should only be able to make a search query after receiving explicit permission from the server). This latter requirement means that the client cannot just download the entire database and run local searches. We present a solution whereby the client downloads the database in *encrypted* form, and then a smartcard is used to carry out a search on the database by enabling the client to decrypt a single database record.

We now provide an inaccurate description of our solution. Denote the ith database record by (p_i, x_i), where p_i is the value of the search attribute (as is standard, the values $p_1, \ldots, p_N$ are unique). We assume that each $p_i \in \{0,1\}^n$, and for some ℓ each $x_i \in \{0,1\}^{\ell n}$ (recall that the pseudorandom permutation works over the domain $\{0,1\}^n$; thus p_i is made up of a single "block" and x_i is made up of ℓ blocks). Then, the server chooses a key k and computes $t_i = F_k(p_i)$, $u_i = F_k(t_i)$ and $c_i = E_{u_i}(x_i)$, for every $i = 1, \ldots, N$. The server sends the encrypted database (t_i, c_i) to the client, together with a smartcard SC that has the key k. The key k is also protected by a challenge/response with a key k_{test} that only the server knows; in addition, after passing a challenge/response, the key k can be used only twice (this is achieved by setting the access-granted counter of k to 2; see Section 2). Now, since F is a pseudorandom function, the value t_i reveals nothing about p_i, and the "key" u_i is pseudorandom, implying that c_i is a cryptographically sound (i.e., secure) encryption of x_i, that therefore reveals nothing about x_i. In order to search the database for attribute p, the client obtains a challenge from the smartcard for k_{test} and sends it to the server. If the server agrees that the client can carry out a search, it computes the response and sends it back. The client then computes $t = F_k(p)$ and $u = F_k(t)$ using the smartcard. If there exists an i for which $t = t_i$, then the client decrypts c_i using the key u, obtaining the record x_i as required. Note that the server has no way of knowing the search query of the client. Furthermore, the client cannot carry out the search without explicit approval from the server, and thus the number of searches can be audited and limited (if required for privacy purposes), or a charge can be issued (if a pay-per-search system is in place).

We warn that the above description is not a fully secure solution. To start with, it is possible for a client to use the key k to compute t and t' for two different values p and p'. Although this means that the client will not be able to obtain the corresponding records x and/or x', it does mean that it can see whether the two values p and p' are in the database (something which it is not supposed to be able to do, because just the existence of an identifier in a database can reveal confidential information). We therefore use two different keys k_1 and k_2; k_1 is used to compute t and k_2 is used to compute u. In addition, we don't use u to directly encrypt x and use the smartcard with a third key k_3 (this is needed to enable a formal reduction to the security of the encryption scheme and for obtaining simulatability).

5.1 The Functionality

We begin by describing the ideal functionality for the problem of oblivious database search; the functionality is a *reactive* one where the server P_1 first sends the database to the trusted party, and the client can then carry out searches. We stress that the client can choose its queries adaptively, meaning that it can choose what keywords to search for after it has already received the output from previous queries. However, each query must be explicitly allowed by the server (this allows the server to limit queries or to charge per query). We first present a basic functionality and then a more sophisticated one:

The Oblivious Database Search Functionality $\mathcal{F}_{\text{basicDB}}$

Functionality $\mathcal{F}_{\text{basicDB}}$ works with a server P_1 and a client P_2 as follows (the variable init is initially set to 0):

Initialize: Upon receiving from P_1 a message $(\mathsf{init}, (p_1, x_1), \ldots, (p_N, x_N))$, if $\mathsf{init} = 0$, functionality $\mathcal{F}_{\text{basicDB}}$ sets $\mathsf{init} = 1$, stores all pairs and sends (init, N) to P_2. If $\mathsf{init} = 1$, then $\mathcal{F}_{\text{basicDB}}$ ignores the message.

Search: Upon receiving a message $\mathsf{retrieve}$ from P_2, functionality $\mathcal{F}_{\text{basicDB}}$ checks that $\mathsf{init} = 1$ and if not it returns $\mathsf{notInit}$. Otherwise, it sends $\mathsf{retrieve}$ to P_1. If P_1 replies with allow then $\mathcal{F}_{\text{basicDB}}$ forwards allow to P_2. When P_1 replies with $(\mathsf{retrieve}, p)$, $\mathcal{F}_{\text{basicDB}}$ works as follows:

1. If there exists an i for which $p = p_i$, functionality $\mathcal{F}_{\text{basicDB}}$ sends $(\mathsf{retrieve}, x_i)$ to P_2
2. If there is no such i, then $\mathcal{F}_{\text{basicDB}}$ sends $\mathsf{notFound}$ to P_2.

If P_1 replies with $\mathsf{disallow}$, then $\mathcal{F}_{\text{basicDB}}$ forwards $\mathsf{disallow}$ to P_2.

Figure 1: The basic oblivious database search functionality

The main drawback with $\mathcal{F}_{\text{basicDB}}$ is that the database is completely static and updates cannot be made by the server. We therefore modify $\mathcal{F}_{\text{basicDB}}$ so that *inserts* and *updates* are included. An insert operation adds a new record to the database, while an update operation makes a change to the x portion of an existing record. We stress that in an update, the previous x value is not erased, but rather the new value is concatenated to the old one. We define the functionality in this way because it affords greater efficiency. Recall that in our protocol, the client holds the entire database in encrypted form. Furthermore, the old and new x portions are encrypted with the same key. Thus, if the client does not erase the old encrypted x value, it can decrypt it at the same time that it is able to decrypt the new x value. Another subtlety that arises is that since inserts are carried out over time, and the client receives encrypted records when they are inserted, it is possible for the client to know when a decrypted record was inserted. In order to model this, we include unique identifiers to records; when a record is inserted, the ideal functionality hands the client the identifier of the inserted record. Then, when a search succeeds, the client receives the identifier together with the x portion. This allows the client in the ideal model to track when a record was inserted (of course, without revealing anything about its content). Finally, we remark that our solution does not efficiently support delete commands (this is for the same reason that updates are modeled as concatenations). We therefore include a reset command that deletes all records. This requires the server to re-encrypt the entire database from scratch and send it to the client. Thus, such a command cannot be issued at too frequent intervals. See Figure 2 for the full definition of $\mathcal{F}_{\rm DB}$.

The Oblivious Database Functionality $\mathcal{F}_{\rm DB}$

Functionality $\mathcal{F}_{\rm DB}$ works with a server P_1 and client P_2 as follows (the variable init is initially set to 0):

Insert: Upon receiving a message (insert, p, x) from P_1, functionality $\mathcal{F}_{\rm DB}$ checks that there is no recorded tuple (id_i, p_i, x_i) for which $p = p_i$. If there is such a tuple it ignores the message. Otherwise, it assigns an identifier id to (p, x), sends (insert, id) to P_2, and records the tuple (id, p, x).

Update: Upon receiving a message (update, p, x) from P_1, functionality $\mathcal{F}_{\rm DB}$ checks that there is a recorded tuple (id_i, p_i, x_i) for which $p = p_i$. If there is no such tuple it ignores the message. Otherwise it updates the tuple, by *concatenating* x to x_i.

Retrieve: Upon receiving a query $(\mathsf{retrieve}, p)$ from the client P_2, functionality $\mathcal{F}_{\rm DB}$ sends retrieve to P_1. If P_2 replies with allow then:

1. If there exists a recorded tuple (id_i, p_i, x_i) for which $p = p_i$, then $\mathcal{F}_{\rm DB}$ sends (id_i, x_i) to P_2.
2. If there does not exist such a tuple, then $\mathcal{F}_{\rm DB}$ sends `notFound` to P_2.

Reset: Upon receiving a message reset from P_1, the functionality $\mathcal{F}_{\rm DB}$ sends reset to P_2 and erases all entries.

Figure 2: A more comprehensive database functionality

5.2 A Protocol for Securely Computing $\mathcal{F}_{\rm basicDB}$

We first present a protocol for securely computing the basic functionality $\mathcal{F}_{\rm basicDB}$. Let F be a (efficiently invertible) pseudorandom permutation over $\{0,1\}^n$ with keys that are uniformly chosen from $\{0,1\}^n$. We define a keyed function $\hat{F}$ from $\{0,1\}^n$ to $\{0,1\}^{\ell n}$ by

$$\hat{F}_k(t) = \langle F_k(t+1), F_k(t+2), \ldots, F_k(t+\ell) \rangle$$

where addition is modulo 2^n. We remark that $\hat{F}_k$ is a pseudorandom function when the input t is uniformly distributed (this actually follows directly from the proof of security in counter mode for block ciphers). We assume that all records in the database are exactly of length ℓn (and that this is known); if this is not the case, then padding can be used.

In our protocol, we use a challenge/response mechanism in the smartcard to restrict use of cryptographic keys. For the sake of concreteness, we assume that the response to a challenge chall with key k_{test} is $F_{k_{\text{test}}}(\mathsf{chall})$ where F is a pseudorandom permutation as above. This makes no difference, and we define it this way for the sake of concreteness only.

Protocol 4 (oblivious database search – basic functionality $\mathcal{F}_{\text{basicDB}}$)

- **Smartcard initialization:** *Party P_1 chooses three keys $k_1, k_2, k_3 \leftarrow \{0,1\}^n$ and imports them into a smartcard SC for use for a pseudorandom permutation. In addition, P_1 imports a key k_{test} as a test object that protects them all by challenge/response. Finally, P_1 sets the access-granted counter of k_1 and k_2 to 1, denoted respectively by $\mathsf{AG}_1, \mathsf{AG}_2$, (and sets no access-granted counter of k_3). See Section 2 for the definition of an access-granted counter.*

  P_1 sends SC to P_2 (this takes place before the protocol below begins). Upon receiving SC, party P_2 checks that there exist three keys with the properties defined above; if not it outputs $\perp$ and halts.[3]

- **The protocol:**
  - **Initialize:** *Upon input $(\mathsf{init}, (p_1, x_1), \ldots, (p_N, x_N))$ for party P_1, the parties work as follows:*
    1. *P_1 randomly permutes the pairs (p_i, x_i).*
    2. *For every i, P_1 computes $t_i = F_{k_1}(p_i)$, $u_i = F_{k_2}(t_i)$ and $c_i = \hat{F}_{k_3}(t_i) \oplus x_i$.*
    3. *P_1 sends $(u_1, c_1), \ldots, (u_N, c_N)$ to P_2 (these pairs are an encrypted version of the database).*
    4. *Upon receiving $(u_1, c_1), \ldots, (u_N, c_N)$, party P_2 stores the pairs and outputs (init, N).*
  - **Search:** *Upon input $(\mathsf{retrieve}, p)$ for party P_2, the parties work as follows:*
    1. *P_2 queries SC for a challenge, receiving chall. P_2 sends chall to P_1.*
    2. *Upon receiving chall, if party P_1 allows the search it computes $\mathsf{resp} = F_{k_{\text{test}}}(\mathsf{chall})$ and sends resp to P_2. Otherwise, it sends $\mathsf{disallow}$ to P_2.*
    3. *Upon receiving resp, party P_2 hands it to SC in order to pass the test. Then:*
       (a) *P_2 uses SC to compute $t = F_{k_1}(p)$ and $u = F_{k_2}(t)$.*
       (b) *If there does not exist any i for which $u = u_i$, then P_2 outputs $\mathsf{notFound}$.*
       (c) *If there exist an i for which $u = u_i$, party P_2 uses SC to compute $r = \hat{F}_{k_3}(t)$; this involves ℓ calls to F_{k_3} in SC. Then, P_2 sets $x = r \oplus c_i$ and outputs $(\mathsf{retrieve}, x)$.*

Theorem 5 *Assume that F is a strong pseudorandom permutation over $\{0,1\}^n$ and let $\hat{F}$ be as defined above. Then, Protocol 4 securely computes $\mathcal{F}_{\text{basicDB}}$.*

Proof: We treat each corruption case separately:

No parties are corrupted. In this case, all the adversary sees within the initialization phase is a list of pairs $(u_1, c_1), \ldots, (u_N, c_N)$ which reveals nothing about the values $(p_1, x_1), \ldots, (p_N, x_N))$ by the fact that F is a pseudorandom permutation. Furthermore, during every search query the adversary sees the values chall and $F_{k_{\text{test}}}(\mathsf{chall})$ which does not give it any useful information about the parties' inputs since k_{test} is chosen independently of k_1, k_2 and k_3.

[3]Not all smartcards allow checking the properties of keys. If not, this will be discovered the first time a search is carried out and then P_2 can just abort then.

Party P_1 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_1; we construct a simulator $\mathcal{S}$ that works as follows:

1. $\mathcal{S}$ obtains the keys k_1, k_2, k_3 that $\mathcal{A}$ imports to the smartcard, as well as the test key k_{test}. If $\mathcal{A}$ does not configure the smartcard correctly, then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\text{basicDB}}$.

2. Upon receiving $(u_1, c_1), \dots, (u_N, c_N)$ from $\mathcal{A}$, simulator $\mathcal{S}$ computes $t_i = F_{k_2}^{-1}(u_i)$, $p_i = F_{k_1}^{-1}(t_i)$ and $x_i = \hat{F}_{k_3}(t_i) \oplus c_i$, for every i. Then, $\mathcal{S}$ sends $(\mathsf{init}, (p_1, x_1), \dots, (p_N, x_N))$ to $\mathcal{F}_{\text{basicDB}}$.

3. Upon receiving a message $\mathsf{retrieve}$ from $\mathcal{F}_{\text{basicDB}}$, simulator $\mathcal{S}$ chooses a random challenge $\mathsf{chall} \in_R \{0,1\}^n$ and hands it to $\mathcal{A}$. Let resp be the response from $\mathcal{A}$. If $\mathsf{resp} = F_{k_{\text{test}}}(\mathsf{chall})$ then $\mathcal{S}$ sends allow to $\mathcal{F}_{\text{basicDB}}$; otherwise, including the case that $\mathcal{A}$ does not respond at all, $\mathcal{S}$ sends $\mathsf{disallow}$.

This completes the simulation. The output distribution from the simulation is identical to a real execution. This is due to the fact that F is a pseudorandom permutation and thus k_1, k_2, k_3 together with a pair (u_i, c_i) define a unique (p_i, x_i) that is sent to $\mathcal{F}_{\text{basicDB}}$. In addition, P_2 can carry out a search if and only if resp is correctly computed; thus, $\mathcal{S}$ sends allow to $\mathcal{F}_{\text{basicDB}}$ if and only if P_2 can carry out a search. Finally, we note that $\mathcal{A}$'s view is identical in the simulation and in a real execution because the only values it sees in both cases are truly random challenges $\mathsf{chall} \in_R \{0,1\}^n$.

Party P_2 is corrupted. We now proceed to the case that P_2 is corrupted. Again, let $\mathcal{A}$ be an adversary controlling P_2; we construct $\mathcal{S}$ as follows:

1. Upon receiving input (init, N) from $\mathcal{F}_{\text{basicDB}}$, simulator $\mathcal{S}$ constructs N tuples $(t_1, u_1, c_1), \dots, (t_N, u_N, c_N)$ where each $t_i, u_i \in_R \{0,1\}^n$ and $c_i \in_R \{0,1\}^{\ell n}$ (recall that ℓ is known to $\mathcal{S}$). $\mathcal{S}$ also chooses $k_{\text{test}} \in_R \{0,1\}^n$. If there exist $i \neq j$ such that $t_i \in \{t_j + 1, \dots, t_j + \ell\}$ or $t_j \in \{t_i + 1, \dots, t_i + \ell\}$, then $\mathcal{S}$ outputs fail_1 and halts.

   $\mathcal{S}$ hands $\mathcal{A}$ the pairs $(u_1, c_1), \dots, (u_N, c_N)$.

2. Upon receiving chall from $\mathcal{A}$, simulator $\mathcal{S}$ sends $\mathsf{retrieve}$ to $\mathcal{F}_{\text{basicDB}}$. If it receives back allow then it computes $\mathsf{resp} = F_{k_{\text{test}}}(\mathsf{chall})$ and hands it to $\mathcal{A}$; if it receives back $\mathsf{disallow}$ then it hands $\mathsf{disallow}$ to $\mathcal{A}$. $\mathcal{S}$ then sets the variables $\mathsf{AG}_1 = \mathsf{AG}_2 = 1$ (these are recordings of the current access granted values).

3. When $\mathcal{A}$ queries F_{k_1} on SC with p, simulator $\mathcal{S}$ checks that $\mathsf{AG}_1 = 1$. If no, it simulates an error message from SC back to $\mathcal{A}$. If yes, it sets $\mathsf{AG}_1 = 0$ and sends $(\mathsf{retrieve}, p)$ to $\mathcal{F}_{\text{basicDB}}$.

   (a) If this is the first time that $\mathcal{A}$ has queried p, then:

      i. If $\mathcal{F}_{\text{basicDB}}$ replies with $\mathsf{notFound}$, then $\mathcal{S}$ chooses a random $t_p \in_R \{0,1\}^n$, stores the pair (p, t_p), and hands t_p to $\mathcal{A}$.

      ii. If $\mathcal{F}_{\text{basicDB}}$ replies with $(\mathsf{retrieve}, x)$, then $\mathcal{S}$ chooses a random index $i \in \{1, \dots, N\}$ that has not yet been chosen, hands t_i to $\mathcal{A}$, and stores the association (i, p, x).

   (b) If this is not the first time that $\mathcal{A}$ queried p, then $\mathcal{S}$ returns the same reply as last time (either t_p or t_i, appropriately).

4. When $\mathcal{A}$ queries F_{k_2} on SC with some t, simulator $\mathcal{S}$ checks that $\mathsf{AG}_2 = 1$. If no, it simulates an error message from SC back to $\mathcal{A}$. If yes, it sets $\mathsf{AG}_2 = 0$ and works as follows:

   (a) If there exists an i and a tuple (t_i, u_i, c_i) where $t = t_i$, then $\mathcal{S}$ hands $\mathcal{A}$ the value u_i from the tuple (t_i, u_i, c_i).
   (b) If there does not exists such an i, then $\mathcal{S}$ chooses a random $u \in_R \{0,1\}^n$ and hands u to $\mathcal{A}$. $\mathcal{S}$ also stores the pair (t, u) so that if t is queried again, then $\mathcal{S}$ will reply with the same u.

5. When $\mathcal{A}$ queries F_{k_3} on SC with some value t, simulator $\mathcal{S}$ checks if there exists an i and a tuple (t_i, u_i, c_i) where $t = t_i + j$ for some $j \in \{1, \ldots, \ell\}$.
   (a) If no, then $\mathcal{S}$ returns a random value ($\mathcal{S}$ stores a set to maintain consistency, meaning that if in the future the same t' is queried, it returns the same random value).
   (b) If yes, then $\mathcal{S}$ checks that there is a recorded tuple (i, p_i, x_i). If no, $\mathcal{S}$ outputs fail_2. Otherwise, it hands $\mathcal{A}$ the n-bit string $c_i^j \oplus x_i^j$ where c_i^j is the jth n-bit block of c_i, and x_i^j is the jth n-bit block of x_i.

   $\mathcal{S}$ continues as above.

This completes the simulation. We begin by showing that in the simulation, the probability that $\mathcal{S}$ outputs fail_1 or fail_2 is negligible. Regarding fail_1, this follows from the fact that ℓ is polynomial in n, and the values t are chosen randomly within a range of size 2^n. Regarding fail_2, recall that $\mathcal{S}$ outputs fail_2 if $\mathcal{A}$ sends a value $t \in \{t_i + 1, \ldots, t_i + \ell\}$ for some t_i in a tuple (t_i, u_i, c_i) but there is no stored tuple (i, p_i, x_i). Now, if no tuple (i, p_i, x_i) is stored, then this means that $\mathcal{S}$ never gave $\mathcal{A}$ the value t_i from the ith tuple (t_i, u_i, c_i). However, t_i is uniformly distributed and so the probability that $\mathcal{A}$ sends $t \in \{t_i + 1, \ldots, t_i + \ell\}$ is negligible.

Next, consider a modification to Protocol 4 where instead of F_{k_1}, F_{k_2} and F_{k_3}, three truly random permutations H_1, H_2 and H_3 are used instead; denote the modified protocol by π'. It is straightforward to show that the output distribution from π' is computationally indistinguishable from the real protocol. This is due to the fact that the protocol can be implemented using an oracle to a random or pseudorandom permutation. We now claim that conditioned on $\mathcal{S}$ not outputting fail_1 and this same event (of overlapping t_i, t_j series) does not occur in π', the output distribution of $\mathcal{S}$ and an honest P_1 in the ideal model is statistically close to the output distribution of $\mathcal{A}$ and an honest P_1 in an execution of the modified protocol π'. This is due to the fact that $\mathcal{S}$ chooses the (t_i, u_i, c_i) values uniformly at random, exactly like an honest P_1 in π' (where truly random permutations are used to compute these values). Now, since none of the t_i and t_j values in $\mathcal{S}$'s simulation or in π' overlap, the distribution over the values in the simulation is exactly as in the execution of π'. However, a bad event can happen if $\mathcal{A}$ can decrypt a block of some c_i without having queried p_i. However, this is exactly the event that causes fail_2 to occur, and we have already shown that this occurs with at most negligible probability. This completes the proof of security. ■

Composability. As in the protocol for set intersection, our simulators do not rewind $\mathcal{A}$ at all. Therefore, our protocol is secure under concurrent general composition.

Remark – adaptive oblivious transfer. Note that the adaptive k-out-of-n oblivious transfer functionality (meaning, oblivious transfer with adaptive queries), is a special case of oblivious database search (where the keywords are just the indices from 1 to n). Thus we obtain an extraordinarily efficient protocol for this problem.

5.3 A Protocol for Securely Computing $\mathcal{F}_{\mathrm{DB}}$

A protocol for securely computing the more sophisticated functionality $\mathcal{F}_{\mathrm{DB}}$ can be derived directly from Protocol 4. Specifically, instead of sending all the pairs (u_i, c_i) at the onset, P_1 sends a new pair every time an insert is carried out. In addition, an update just involves P_1 re-encrypting the new x_i value and sending the new ciphertext c_i'. Finally, a reset is carried out by choosing new keys k_1, k_2, k_3 and writing them to the smartcard (deleting the previous ones). Then, any future inserts are computed using these new keys. Of course, the new keys are written to the smartcard using secure messaging, as we have described above.

6 Oblivious Document Search

In Section 5 we showed how a database can be searched obliviously, where the search is based only on a key attribute. Here, we show how to extend this to a less structured database, and in particular to a corpus of texts. In this case, there are many keywords that are associated with each document and the user wishes to gain access to all of the documents that contain a specific keyword. A naive solution would be to define each record value so that it contains all the documents which the keyword appears in. However, this would be horrifically expensive because the same document would have to be repeated many times. We present a solution where each document is stored (encrypted) only once, as follows.

Our solution uses Protocol 4 as a subprotocol, and we model this by constructing our protocol for oblivious document search in a "hybrid" model where a trusted party is used to compute the ideal functionality $\mathcal{F}_{\mathrm{basicDB}}$. (The soundness of working in this way was proven in [5].) The basic idea is for the parties to use $\mathcal{F}_{\mathrm{basicDB}}$ to store an index to the corpus of texts as follows. The server chooses a random value s_i for every document D_i and then associates with a keyword p the values s_i where p appears in the document D_i. Then, this index is sent to $\mathcal{F}_{\mathrm{basicDB}}$, enabling P_2 to search it obliviously. In addition, P_1 encrypts document D_i using a smartcard and s_i in the same way that the x_i values are encrypted using t_i in Protocol 4. Since P_2 is only able to decrypt a document if it has the appropriate s_i value, it can only do this if it queried $\mathcal{F}_{\mathrm{basicDB}}$ with a keyword p that is in document D_i. Observe that in this way, each document is only encrypted once.

Let $\mathcal{P}$ be the space of keywords of size M, let $D_1, \ldots, D_N$ denote N text documents, and let $P_i = \{p_{i_j}\}$ be the set of keywords that appear in D_i (note $P_i \subseteq \mathcal{P}$). Using this notation, when a search is carried out for a keyword p, the client is supposed to receive the set of documents D_i for which $p \in P_i$. We now proceed to formally define the oblivious document search functionality $\mathcal{F}_{\mathrm{doc}}$.

Our protocol uses an additional tool of perfectly-hiding commitment scheme denoted by $(\mathsf{com}, \mathsf{dec})$ that enables a party to commit to a value while keeping it secret (even from all powerful adversary); see [12] for a formal definition. We let $\mathsf{com}(m; r)$ denotes the commitment to a message m using random coins r. For efficiency, we instantiate $\mathsf{com}(\cdot;\cdot)$ with Pedersen's commitment scheme [22]. Assume, for simplicity, that $q - 1 = 2q'$ for some prime q', and let g, h be generators of a subgroup of $\mathbb{Z}_q^*$ of order q'. A commitment to m is then defined as $\mathsf{com}(m; r) = g^m h^r$ where $r \leftarrow_R \mathbb{Z}_{q-1}$. The scheme is perfectly hiding as for every m, r, m' there exists r' such that $g^m h^r = g^{m'} h^{r'}$. The scheme is binding assuming hardness of computing $\log_g h$.

The Oblivious Document Search Functionality $\mathcal{F}_{\text{doc}}$

Functionality $\mathcal{F}_{\text{doc}}$ works with a server P_1 and client P_2 as follows (the variable init is initially set to 0):

Initialize: Upon receiving from P_1 a message $(\mathsf{init}, \mathcal{P}, D_1, \ldots, D_N)$, if $\mathsf{init} = 0$, functionality $\mathcal{F}_{\text{doc}}$ sets $\mathsf{init} = 1$, stores all documents and $\mathcal{P}$, and sends (init, N, M) to P_2, where N is the number of documents and M is the size of the keyword set M. If $\mathsf{init} = 1$, then $\mathcal{F}_{\text{doc}}$ ignores the message.

Search: Upon receiving a message search from P_2, functionality $\mathcal{F}_{\text{doc}}$ checks that $\mathsf{init} = 1$ and if not it returns $\mathsf{notInit}$. Otherwise, it sends search to P_1. If P_1 replies with allow then $\mathcal{F}_{\text{doc}}$ forwards allow to P_2. When P_2 replies with (search, p), $\mathcal{F}_{\text{doc}}$ works as follows:

1. If there exists an i for which $p \subseteq P_i$, functionality $\mathcal{F}_{\text{doc}}$ sends $(\mathsf{search}, \{D_i\}_{p \in P_i})$ to P_2.
2. If there is no such i, then $\mathcal{F}_{\text{doc}}$ sends $\mathsf{notFound}$ to P_2.

If P_1 replies with $\mathsf{disallow}$, then $\mathcal{F}_{\text{doc}}$ forwards $\mathsf{disallow}$ to P_2.

Figure 3: Oblivious document search via keywords

We now present the protocol for securely computing $\mathcal{F}_{\text{doc}}$. Recall that our protocol uses a trusted party to compute $\mathcal{F}_{\text{basicDB}}$. Of course, the real protocol uses Protocol 4 as a subprotocol; the presentation using $\mathcal{F}_{\text{basicDB}}$ is simply clearer.

Protocol 6 (oblivious document search by keyword)

- **Smartcard initialization:** *Party P_1 chooses a key $k \leftarrow \{0,1\}^n$ and imports it into a smartcard SC for use for a pseudorandom permutation. P_1 sends SC to P_2 (this takes place before the protocol below begins).*
- **The protocol:**
  - **Initialize:** *Upon input $(\mathsf{init}, \mathcal{P}, D_1, \ldots, D_N)$ to P_1, the parties work as follows:*
    1. *The server P_1 initializes a smartcard with a key k for a pseudorandom permutation, and sends the smartcard to P_2.*
    2. *P_1 chooses random values $s_1, \ldots, s_N \in_R \{0,1\}^n$ (one random value for each document), and sends P_2 the commitments $\{\mathsf{com}_i = \mathsf{com}(s_i; r_i)\}_{i=1}^N$ where $r_1, \ldots, r_N$ are random strings of appropriate length.*
    3. *Then, P_1 defines a database of M records (p_j, x_j) where $p_j \in \mathcal{P}$ is a keyword, and $x_j = \{(i, (s_i, r_i))\}_{p_j \in D_i}$ (i.e., x_j is the set of pairs $(i, (s_i, r_i))$ where i is such that p_j appears in document D_i). Finally, it encrypts each document D_i by computing $C_i = \hat{F}_k(s_i) \oplus D_i$ (see Section 5.2 for the definition of $\hat{F}$).*
    4. *P_1 sends $C_1, \ldots, C_N$ to P_2, and sends $(\mathsf{init}, (p_1, x_1), \ldots, (p_M, x_M))$ to $\mathcal{F}_{\text{basicDB}}$.*
    5. *Upon receiving $\mathsf{com}_1, \ldots, \mathsf{com}_N$ and $C_1, \ldots, C_N$ from P_1 and (init, M) from $\mathcal{F}_{\text{basicDB}}$, party P_2 outputs (init, N, M).*
  - **Search:** *Upon input (search, p) to P_2, the parties work as follows:*
    1. *The client P_2 sends $(\mathsf{retrieve}, p)$ to $\mathcal{F}_{\text{basicDB}}$ and receives back a set $x = \{(i, (s_i, r_i))\}$.*
    2. *For every i in the set x, party P_2 verifies first that $\mathsf{com}_i = \mathsf{com}(s_i, r_i)$. If the verification holds it uses the smartcard to compute $D_i = \hat{F}_k(s_i) \oplus C_i$.*
    3. *P_2 outputs $(\mathsf{search}, \{D_i\})$ where $\{D_i\}$ is the set of documents obtained above.*

We have the following theorem, that can be derived from the proof of Theorem 5.

Theorem 7 *Assume that F is a pseudorandom permutation over $\{0,1\}^n$ and let $\hat{F}$ be as defined in Section 5.2. Then, Protocol 6 securely computes $\mathcal{F}_{\rm doc}$ when Protocol 4 is used in place of the trusted party computing $\mathcal{F}_{\rm basicDB}$.*

Proof: We treat each corruption case separately. Our proof is in a hybrid model where a trusted party computes an ideal functionality $\mathcal{F}_{\rm basicDB}$.

No parties are corrupted. In this case, all the adversary sees are the sets $\mathsf{com}_1, \ldots, \mathsf{com}_N$ and $C_1, \ldots, C_N$ which reveal nothing about the values $(\mathcal{P}, D_1, \ldots, D_N)$ by the facts that com is a perfectly hiding commitment scheme and F is a pseudorandom permutation (recall that the rest of the messages are sent via the ideal execution of $\mathcal{F}_{\rm basicDB}$).

Party P_1 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_1; we construct a simulator $\mathcal{S}$ that works as follows:

1. $\mathcal{S}$ obtains the keys k that $\mathcal{A}$ imports to the smartcard. If $\mathcal{A}$ does not configure the smartcard correctly, then $\mathcal{S}$ sends $\perp$ to $\mathcal{F}_{\rm doc}$.
2. Upon receiving from $\mathcal{A}$, $(\mathsf{com}_1, \ldots, \mathsf{com}_N)$, $(C_1, \ldots, C_N)$ and $(\mathsf{init}, (p_1, x_1), \ldots, (p_M, x_M))$ (where the last message is being sent to $\mathcal{F}_{\rm basicDB}$), simulator $\mathcal{S}$ sets $\mathcal{P} = \{p_1, \ldots, p_M\}$. Then, $\mathcal{S}$ computes $D_i = \hat{F}_k(s_i) \oplus C_i$ only if there exists a pair (i,j) for which $(i,(s_i,r_i)) \in x_j$ and $\mathsf{com}_i = \mathsf{com}(s_i;r_i)$. Now, for every p_j, if there exists i such that $(i,(s_i,r_i)) \in x_j$ yet $\mathsf{com}_i \neq \mathsf{com}(s_i;r_i)$ and in addition, $\mathcal{S}$ recorded D_i above, $\mathcal{S}$ deletes p_j from D_i. $\mathcal{S}$ completes its set of documents (if needed) to size N by adding random documents of appropriate size. Finally, $\mathcal{S}$ sends $(\mathsf{init}, \mathcal{P}, D_1, \ldots, D_N)$ to $\mathcal{F}_{\rm doc}$.
3. Upon receiving search from $\mathcal{F}_{\rm doc}$, $\mathcal{S}$ hands $\mathcal{A}$ the message $\mathsf{retrieve}$ and forwards $\mathcal{F}_{\rm basicDB}$ $\mathcal{A}$'s response.

This completes the simulation. We prove that the output distribution from the simulation is computationally indistinguishable from the real execution as follows. Let fail_1 denotes the event for which there exists i, j_0, j_1 such that $(i,(s_{i_b}, r_{i_b})) \in x_{j_b}$ and $\mathsf{com}_i = \mathsf{com}(s_{i_b}, r_{i_b})$ for both $b \in \{0,1\}$. Note that if fail_1 occurs the simulation fails since the real P_2 cannot conclusively compute D_i. This is in contrast to the simulation where P_2 outputs a unique value of D_i. Clearly, the probability that fail_1 occurs is negligible due to the security of com. We further denote the event fail_2 for which $\mathcal{S}$ completes its set of documents in Step 2 of the simulation with a document D that contains some keyword $p \in \mathcal{P}$. Note that the simulation fails if P_2 queries $\mathcal{S}$ on p. However due to the fact that $\mathcal{S}$ chooses random documents fail_2 is only likely to occur with negligible probability.

Now, conditioning on $\overline{\mathsf{fail}_1}$ and $\overline{\mathsf{fail}_2}$ we have that P_2 outputs the exact same value in these executions since it ignores every document D_i for which it does not receive a valid decommitment for com_i. Specifically for every query p_j of P_2, it only outputs D_i such that $(i,(s_i,r_i)) \in x_j$ and $\mathsf{com}_i = \mathsf{com}(s_i, r_i)$, exactly as in the simulation.

Part P_2 is corrupted. Let $\mathcal{A}$ be an adversary controlling P_2; we construct a simulator $\mathcal{S}$ that works as follows:

1. Upon receiving (init, N, M) from $\mathcal{F}_{\text{doc}}$, simulator $\mathcal{S}$ chooses N random pairs (s_i, r_i) of appropriate length. If there exist $i \neq j$ such that $s_i \in \{s_j+1, \ldots, s_j+\ell\}$ or $s_j \in \{s_i+1, \ldots, s_i+\ell\}$, then $\mathcal{S}$ outputs fail_1 and halts. Otherwise it sends $\mathcal{A}$ the commitments $\mathsf{com}_i = \mathsf{com}(s_i; r_i)$.
2. $\mathcal{S}$ also chooses N random strings $C_1, \ldots, C_N$ and hands them to $\mathcal{A}$.
3. $\mathcal{S}$ emulate $\mathcal{F}_{\text{basicDB}}$ and receives from $\mathcal{A}$ the message $(\mathsf{retrieve}, p)$. It then sends search to its trusted party that computes $\mathcal{F}_{\text{doc}}$. If $\mathcal{F}_{\text{doc}}$ responds with allow $\mathcal{S}$ sends it (search, p). Otherwise it hands $\mathcal{A}$ $\mathsf{disallow}$.
   (a) If $\mathcal{F}_{\text{doc}}$ returns documents $D_1, \ldots, D_\ell$, $\mathcal{S}$ continues as follow. It chooses ℓ random indices $i_1, \ldots, i_\ell \in \{1, \ldots, N\}$ and sets $x = \{(i', (s_{i'}, r_{i'}))\}$ for all $i' \in \{i_1, \ldots, i_\ell\}$. It then sends x to $\mathcal{A}$, emulating $\mathcal{F}_{\text{basicDB}}$.
   (b) If $\mathcal{F}_{\text{doc}}$ returns $\mathsf{notFound}$, $\mathcal{S}$ forwards it to $\mathcal{A}$.
4. When $\mathcal{A}$ queries F_k on SC with some s, simulator $\mathcal{S}$ works as follows:
   (a) If there exist an $i' \in \{i_1, \ldots, i_\ell\}$ and an α where $s = s_{i'} + \alpha$, then $\mathcal{S}$ hands $\mathcal{A}$ the αth n-bit block of $C_{i'} \oplus D_{i'}$.
   (b) If there exist an $i' \notin \{i_1, \ldots, i_\ell\}$ and an α where $s = s_{i'} + \alpha$, $\mathcal{S}$ outputs fail_2.
   (c) Otherwise $\mathcal{S}$ chooses a random $u \in_R \{0,1\}^n$ and hands u to $\mathcal{A}$. $\mathcal{S}$ also stores the pair (s, u) so that if s is queried again, then $\mathcal{S}$ will reply with the same u.

This completes the simulation. Note first that the probability that $\mathcal{S}$ outputs fail_1 or fail_2 is negligible by applying the same arguments from the previous proof. Now, recall that the only two messages that $\mathcal{A}$ sees are $(\mathsf{com}_1, \ldots, \mathsf{com}_N)$, which distributes identically in both executions due to the hiding property of com and $(C_1, \ldots, C_N)$. We further claim that the output distribution of $\mathcal{S}$ and an honest P_1 in the ideal model and the output distribution of $\mathcal{A}$ and an honest P_1 in an execution of Protocol 6 is computationally indistinguishable.

Consider a modification to Protocol 6 where instead of F_k, a truly random permutation H is used instead; denote the modified protocol by π'. It is straightforward to show that the output distribution from π' is computationally indistinguishable from the hybrid protocol. This is due to the fact that the protocol can be implemented using an oracle to a random or pseudorandom permutation. We now claim that conditioned on $\mathcal{S}$ not outputting fail_1 and the analogous event (of s_i, s_j overlapping) does not occur in π', the output distribution of $\mathcal{S}$ and an honest P_1 in the ideal model is statistically close to the output distribution of $\mathcal{A}$ and an honest P_1 in an execution of the modified protocol π'. This is due to the fact that $\mathcal{S}$ chooses the u values uniformly at random, exactly like an honest P_1 in π' (where truly random permutations are used to compute these values). Now, since none of the s_i and s_j values in $\mathcal{S}$'s simulation or in π' overlap, the distribution over the values is exactly as in the execution of π'. However, a bad event can happen if $\mathcal{A}$ can decrypt a block of some C_i without learning s_i. However, this is exactly the event that causes fail_2 to occur, and we have already shown that this occurs with at most negligible probability. This completes the proof of security. ■

7 Conclusions and Future Directions

We have shown that standard smartcards and standard smartcard infrastructure can be used to construct secure protocols that are orders of magnitude more efficient than all previously known solutions. In addition to being efficient enough to be used in practice, our protocols have full proofs of security and achieve simulation according to the ideal/real model paradigm. No cryptographic protocol for a realistic model has achieved close to the level of efficiency of our protocols. Finally, we note that since standard smartcards are used, it is not difficult to deploy our solutions in practice (especially given the fact that smartcards are become more and more ubiquitous today).

We believe that this model should be studied further with the aim of bridging the theory and practice of secure protocols. In addition to studying what can be achieved in the preferred setting where only standard smartcards are used, it is also of interest to construct highly efficient protocols that use special-purpose smartcards that can be implemented in Java applets on Javacards.

Acknowledgements

We thank Danny Tabak for programming the demo of the set intersection protocol.

References

[1] G. Aggarwal, N. Mishra and B. Pinkas. Secure Computation of the K'th-ranked Element. In *EUROCRYPT 2004*, Springer-Verlag (LNCS 3027), pages 40–55, 2004.

[2] Y. Aumann and Y. Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In 4*th TCC*, Springer-Verlag (LNCS 4392), pages 137-156, 2007.

[3] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[4] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] R. Canetti, Y. Ishai, R. Kumar, M.K. Reiter, R. Rubinfeld and R. Wright. Selective Private Function Evaluation with Applications to Private Statistics. In 20*th PODC*, pages 293–304, 2001.

[7] D. Chaum, C. Crépeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[8] B. Chor, N. Gilboa, and M. Naor. Private Information Retrieval by Keywords. *Technical Report TR-CS0917,* Department of Computer Science, Technion, 1997.

[9] B. Chor, O. Goldreich, E. Kushilevitz and M. Sudan. Private Information Retrieval. *Journal of the ACM,* 45(6):965–981, 1998.

[10] M.J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *TCC 2005*, Springer-Verlag (LNCS 3378), pages 303–324, 2005.

[11] M.J. Freedman, K. Nissim and B. Pinkas. Efficient Private Matching and Set Intersection. In *EUROCRYPT 2004*, Springer-Verlag (LNCS 3027), pages 1–19, 2004.

[12] O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools.* Cambridge University Press, 2001.

[13] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

[14] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

[15] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[16] C. Hazay and Y. Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In 5*th TCC*, Springer-Verlag (LNCS 4948), pages 155–175, 2008.

[17] L. Kissner and D.X. Song. Privacy-Preserving Set Operations. In *CRYPTO 2005*, Springer-Verlag (LNCS 3621), pages 241–257, 2005.

[18] E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In 38*th STOC,* pages 109–18, 2006.

[19] Y. Lindell and B. Pinkas. Privacy Preserving Data Mining. *Journal of Cryptology*, 15(3):177–206, 2002. An extended abstract appeared in *CRYPTO 2000.*

[20] S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[21] M. Naor and B. Pinkas. Oblivious Transfer and Polynomial Evaluation. In 31*st STOC*, pages 245–254, 1999.

[22] T. P. Pedersen. Non-Interactive and Information-Theoretical Secure Verifiable Secret Sharing. *CRYPTO 1991*, Springer-Verlag (LNCS 576) pages 129–140, 1991.

[23] M. Witteman. Advances in Smartcard Security. *Information Security Bulletin*, July 2002, pages 11–22, 2002.

[24] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.