Private Searching On Streaming Data[⋆]

Rafail Ostrovsky [1⋆⋆] and William E. Skeith III [2]

[1] UCLA Computer Science Department, Email: rafail@cs.ucla.edu
[2] UCLA Department of Mathematics, Email: wskeith@math.ucla.edu

Abstract. In this paper, we consider the problem of private searching on streaming data. We show that in this model we can efficiently implement searching for documents under a secret criteria (such as presence or absence of a hidden combination of hidden keywords) under various cryptographic assumptions. Our results can be viewed in a variety of ways: as a generalization of the notion of a Private Information Retrieval (to the more general queries and to a streaming environment as well as to public-key program obfuscation); as positive results on privacy-preserving datamining; and as a delegation of hidden program computation to other machines.

KEYWORDS Code Obfuscation, Crypto-computing, Software security, Database security, Public-key Encryption with special properties, Private Information Retrieval, Privacy-Preserving Keyword Search, Secure Algorithms for Streaming Data, Privacy-Preserving Datamining, Secure Delegation of Computation, Searching with Privacy, Mobile code.

1 Introduction

DATA FILTERING FOR THE INTELLIGENCE COMMUNITY. As our motivating example, we examine one of the tasks of the intelligence community: to collect "potentially useful" information from huge streaming sources of data. The data sources are vast, and it is often impractical to keep all the data. Thus, streaming data is typically sieved from multiple data streams in an on-line fashion, one document/message/packet at a time, where most of the data is immediately dismissed and dropped to the ground, and only some small fraction of "potentially useful" data is retained. These streaming data sources, just to give a few examples, include things like packet traffic on some network routers, on-line news feeds (such as Reuters.com), some internet chat-rooms, or potentially terrorist-related blogs or web-cites. Of course, most of the data is totally innocent and is immediately dismissed except for some data that raises "red flags" is collected for later analysis "on the inside".

⋆ For the full version of this paper, see the IACR E-Print Archive.

⋆⋆ Supported in part by Intel equipment grant, Teradata Corporation, NSF Cybertrust grant, OKAWA research award, B. John Garrick Foundation and Xerox Innovation group Award.

In almost all cases, what's "potentially useful" and raises a "red flag" is classified, and satisfies a secret criteria (i.e., a boolean decision whether to keep this document or throw it away). The classified "sieving" algorithm is typically written by various intelligence community analysts. Keeping this "sieving" algorithm classified is clearly essential, since otherwise adversaries could easily avoid their messages from being collected by simply avoiding criteria that is used to collect such documents in the first place. In order to keep the selection criteria classified, one possible solution (and in fact the one that is used in practice) is to collect *all* streaming data "on the inside" —in a secure environment— and then filter the information, according to classified rules, throwing away most of it and keeping only a small fraction of data-items that are interesting according to the secret criteria, such as a set of keywords that raise a red-flag. While this certainly keeps the sieving information private, this solution requires **transferring** all streaming data to a classified environment, adding considerable cost, both in terms of communication cost and a potential delay or even loss of data, if the transfer to the classified environment is interrupted or dropped in transit. Furthermore, it requires considerable cost of **storage** of this (un-sieved) data in case the transfer to the classified setting is delayed.

Clearly, a far more preferable solution, is to sieve all these data-streams at their sources (even on the same computers or routers where the stream is generated or arrives in the first place). The issue, of course, is how can we do it while keeping sieving criteria classified, even if the computer where the sieving program executes falls into enemy's hands? Perhaps somewhat surprisingly, we show how to do just that while keeping the sieving criteria provably hidden from the adversary, even if the adversary gets to experiment with the sieving executable code and/or tries to reverse-engineer it. Put differently, we construct a "compiler" (i.e. of how to compile sieving rules) so that it is provably impossible to reverse-engineer the classified rules from the executable complied sieving code. Now, we state our results in a more general terms, that we believe are of independent interest:

PUBLIC-KEY PROGRAM OBFUSCATION: Very informally, given a program f from a complexity class $\mathcal{C}$, and a security parameter k, a **public-key program obfuscator** compiles f into (F, Dec), where F on any input computes an encryption of what f would compute on the same input. The decryption algorithm Dec decrypts the output of F. That is, for any input x, $Dec(F(x)) = f(x)$, but given code for F it is impossible to distinguish for any polynomial time adversary which f from complexity class $\mathcal{C}$ was used to produce F . We stress that in our definition, the program encoding length $|F|$ must polynomially depend only on $|f|$ and k, and the output length of $|F(x)|$ must polynomially depend only on $|f(x)|$ and k. It is easy to see that Single-Database Private Information Retrieval (including keyword search) can be viewed as a special case of public-key program obfuscator.

OBFUSCATING SEARCHING ON STREAMING DATA: We consider how to public-key program obfuscate Keyword Search algorithms on streaming data, where the size of the query (i.e. complied executable) must be *independent* of the size of stream (i.e., database), and that can be executed in an on-line environment, one document at a time. Our results also can be viewed as improvement and a speedup of the best previous results of single-round PIR with keyword search of Freedman, Ishai, Pinkas and Reingold [10]. In addition to the introduction of the streaming model, this paper also improves the previous work on keyword PIR by allowing for the simultaneous return of multiple documents that match a set of keywords, and also the ability to more efficiently perform different types of queries beyond just searching for a single keyword. For example, we show how to search for the disjunction of a set of keywords and several other functions.

OUR RESULTS: We consider a dictionary of finite size (e.g., an English dictionary) D that serves as the universe for our keywords. Additionally, we can also have keywords that must be absent from the document in order to match it. We describe the various properties of such filtering software below. A filtering program F stores up to some maximum number m of matching documents in an encrypted buffer B. We provide several methods for constructing such software F that saves up to m matching documents with overwhelming probability and saves non-matching documents with negligible probability (in most cases, this probability will be identically 0), all without F or its encrypted buffer B revealing any information about the query that F performs. The requirement that non-matching documents are not saved (or at worst with negligible probability) is motivated by the streaming model: in general the number of non-matching documents will be vast in comparison to those that do match, and hence, to use only small storage, we must guarantee that non-matching documents from the stream do not collect in our buffer. Among our results, we show how to execute queries that search for documents that match keywords in a disjunctive manner, i.e., queries that search for documents containing one or more keywords from a keyword set. Based on the Paillier cryptosystem, [18], we provide a construction where the filtering software F runs in $O(l \cdot k^3)$ time to process a document, where k is a security parameter, and l is the length of a document, and stores results in a buffer bounded by $\mathcal{O}(m \cdot l \cdot k^2)$. We stress again that F processes documents one at a time in an online, streaming environment. The size of F in this case will be $O(k \cdot |D|)$ where $|D|$ is the size of the dictionary in words. Note that in the above construction, the program size is proportional to the dictionary size. We can in fact improve this as well: we have constructed a reduced program size model that depends on the *Φ-Hiding Assumption* [5]. The running time of the filtering software in this implementation is linear in the document size and is $O(k^3)$ in the security parameter k. The program size for this model is only $O(polylog(|D|))$. We also have an abstract construction based on any group homomorphic, semantically secure encryption scheme. Its performance depends on the performance of the underlying encryption scheme, but will generally perform similarly to the above constructions. As mentioned above, all of these constructions have size that is independent of the size of the data stream. Also, using the results of Boneh, Goh, and Nissim [2], we show how to execute queries that search for an "AND" of two sets of keywords (i.e., the query searches for documents that contain at least one word from K_1 *and* at least one word from K_2 for sets of keywords K_1, K_2), without asymptotically increasing the program size.

Our contributions can be divided into three major areas: Introduction of the streaming model; having queries simultaneously return multiple results; and the ability to extend the semantics of queries beyond just matching a single keyword.

COMPARISON WITH PREVIOUS WORK: A superficially related topic is that of "searching on encrypted data" (e.g., see [3] and the references therein). We note that this body of work is in fact not directly relevant, as the data (i.e. input stream) that is being searched is not encrypted in our setting.

The notion of obfuscation was considered by [1], but we stress that our setting is different, since our notion of public-key obfuscation allows the output to be encrypted, whereas the definition of [1] demands the output of the obfuscated code is given in the clear, making the original notion of obfuscation much more demanding.

Our notion is also superficially related to the notion of "crypto-computing" [19]. However, in this work we are concerned with programs that contain loops, and where we cannot afford to expand this program into circuits, as this will blow-up the program size.

Our work is most closely related to the notion of Single-database Private Information Retrieval (PIR), that was introduced by Kushilevitz and Ostrovsky [13] and has received a lot of subsequent attention in the literature [13, 5, 8, 17, 14, 4, 20, 15, 10]. (In the setting of multiple, non-communicating databases, the PIR notion was introduced in [7].) In particular, the first PIR with poly-logarithmic overhead was shown by Cachin, Micali and Stadler [5], and their construction can be executed in the streaming environment. Thus the results of this paper can be viewed as a generalization of their work as well. The setting of single-database PIR keyword search was considered in [13, 6, 12] and more recently by Freedman, Ishai, Pinkas and Reingold [10]. The issue of multiple matches of a single keyword (in a somewhat different setting) was considered by Kurosawa and Ogata [12].

There are important differences between previous works and our work on single-database PIR keyword search: in the streaming model, the program size must be *independent* of the size of the stream, as the stream is assumed to be an arbitrarily polynomially-large source of data and the complier does not need to know the size of the stream when creating the obfuscated query. In contrast, in all previous non-trivial PIR protocols, when creating the query, the user of the PIR protocol must know the upper bound on the database size while creating the PIR query. Also, as is necessary in the streaming model, the memory needed for our scheme is bounded by a value proportional to the size of a document as well as an upper bound on the total number of documents we wish to collect, but is independent of the size of the stream of documents. Finally, we have also extended the types of queries that can be performed. In previous work on keyword PIR, a single keyword was searched for in a database and a single result returned. If one wanted to query an "OR" of several keywords, this would require creating several PIR queries, and then sending each to the database.

We however show how to intrinsically extend the types of queries that can be performed, without loss of efficiency or with multiple queries. In particular, all of our systems can efficiently perform an "OR" on a set of keywords and its negation (i.e. a document matches if certain keyword is absent from the document). In addition, we show how to privately execute a query that searches for an "AND" of two sets of keywords, meaning that a document will match if and only if it contains at least one word from each of the keyword sets without the increase in program (or dictionary) size.

2 Definitions and Preliminaries

2.1 Basic Definitions

For a set V we denote the power set of V by $\mathcal{P}(V)$.

Definition 1. *Recall that a function $g : \mathbb{N} \to \mathbb{R}^+$ is said to be* negligible *if for any $c \in \mathbb{N}$ there exists $N_c \in \mathbb{Z}$ such that $n \geq N_c \Rightarrow g(n) \leq \frac{1}{n^c}$.*

Definition 2. *Let $\mathcal{C}$ be a class of programs, and let $f \in \mathcal{C}$. We define a* public key program obfuscator in the weak sense *to be an algorithm*

$$\mathsf{Compile}(f, r, 1^k) \mapsto \{F(\cdot,\cdot), \mathsf{Decrypt}(\cdot)\}$$

where r is randomness, k is a security parameter, and F and $\mathsf{Decrypt}$ are algorithms with the following properties:

- *(Correctness) F is a probabilistic function such that $\forall x, \Pr_{R,R'}\Big[\mathsf{Decrypt}(F(x, R')) = f(x)\Big] \geq 1 - neg(k)$*
- *(Compiled Program Conciseness) There exists a constant c such that $|f| \geq \frac{|F(\cdot,\cdot)|}{(|f|+k)^c}$*
- *(Output Conciseness) There exists a constant c such that For all x, R $|f(x)| \geq \frac{|F(x,R)|}{k^c}$*
- *(Privacy) Consider the following game between an adversary A and a challenger C:*
  1. *On input of a security parameter k, A outputs two functions $f_1, f_2 \in \mathcal{C}$.*
  2. *C chooses a $b \in \{0, 1\}$ at random and computes $\mathsf{Compile}(f_b, r, k) = \{F, \mathsf{Decrypt}\}$ and sends F back to A.*
  3. *A outputs a guess b'.*

  We say that the adversary wins if $b' = b$, and we define the adversary's advantage to be $\mathrm{Adv}_A(k) = |\Pr(b = b') - \frac{1}{2}|$. Finally we say that the system is secure if for all $A \in PPT$, $\mathrm{Adv}_A(k)$ is a negligible function in k.

We also define a stronger notion of this functionality, in which the decryption algorithm does not give any information about f beyond what can be learned from the output of the function alone.

Definition 3. *Let $\mathcal{C}$ be a class of programs, and let $f \in \mathcal{C}$. We define a* public key program obfuscator in the strong sense *to be a triple of algorithms* $(\mathsf{Key\text{-}Gen}, \mathsf{Compile}, \mathsf{Decrypt})$ *defined as follows:*

1. $\mathsf{Key\text{-}Gen}(k)$*: Takes a security parameter k and outputs a public key and a secret key $A_{public}, A_{private}$.*
2. $\mathsf{Compile}(f, r, A_{public}, A_{private})$*: Takes a program $f \in \mathcal{C}$, randomness r and the public and private keys, and outputs a program $F(\cdot, \cdot)$ that is subject to the same Correctness and conciseness properties as in Definition 2.*
3. $\mathsf{Decrypt}(F(x), A_{private})$*: Takes output of F and the private key and recovers $f(x)$, just as in the correctness of Definition 2.*

Privacy is also defined as in Definition 2, however in the first step the adversary A receives as an additional input A_{public} and we also require that Decrypt *reveals no information about f beyond what could be computed from $f(x)$: Formally, for all adversaries $A \in PPT$ and for all history functions h there exists a simulating program $B \in PPT$ that on input $f(x)$ and $h(x)$ is computationally indistinguishable from A on input* $(\mathsf{Decrypt}, F(x), h(x))$.

Now, we give instantiations of these general definitions to the class of programs $\mathcal{C}$ that we show how to handle: We consider a universe of words $W = \{0,1\}^*$, and a dictionary $D \subset W$ with $|D| = \alpha < \infty$. We think of a document just to be an ordered, finite sequence of words in W, however, it will often be convenient to look at the set of distinct words in a document, and also to look at some representation of a document as a single string in $\{0,1\}^*$. So, the term *document* will often have several meanings, depending on the context: if M is said to be a *document*, generally this will mean M is an ordered sequence in W, but at times, (e.g., when M appears in set theoretic formulas) *document* will mean (finite) element of $\mathcal{P}(W)$ (or possibly $\mathcal{P}(D)$), and at other times still, (say when one is talking of bit-wise encrypting a document) we'll view M as $M \in \{0,1\}^*$. We define a *set of keywords* to be any subset $K \subset D$. Finally, we define a *stream* of documents S just to be any sequence of documents.

We will consider only a few types of queries in this work, however would like to state our definitions in generality. We think of a *query type*, $\mathcal{Q}$ as a class of logical expressions in $\wedge, \vee$, and $\neg$. For example, $\mathcal{Q}$ could be the class of expressions using only the operation $\wedge$. Given a query type, one can plug in the number of variables, call it α for an expression, and possibly other parameters as well, to select a specific boolean expression, Q in α variables from the class $\mathcal{Q}$. Then, given this logical expression, one can input $K \subset D$ where $K = \{k_i\}_{i=1}^{\alpha}$ and create a function, call it $Q_K : \mathcal{P}(D) \rightarrow \{0,1\}$ that takes documents, and returns 1 if and only if a document matches the criteria. $Q_K(M)$ is computed simply by evaluating Q on inputs of the form $(k_i \in M)$. We will call Q_K a *query over keywords K*.

We note again that our work does not show how to privately execute arbitrary queries, despite the generality of these definitions. In fact, extending the query semantics is an interesting open problem.

Definition 4. *For a query Q_K on a set of keywords K, and for a document M, we say that M* matches *query Q_K if and only if $Q_K(M) = 1$.*

Definition 5. *For a fixed query type $\mathcal{Q}$, a* private filter generator *consists of the following probabilistic polynomial time algorithms:*

1. $\textsf{Key-Gen}(k)$*: Takes a security parameter k and generates public key A_{public}, and a private key $A_{private}$.*
2. $\textsf{Filter-Gen}(D, Q_K, A_{public}, A_{private}, m, \gamma)$*: Takes a dictionary D, a query $Q_K \in \mathcal{Q}$ for the set of keywords K, along with the private key and generates a search program F. F searches any document stream S (processing one document at a time and updating B) collects up to m documents that match Q_K in B, outputting an encrypted buffer B that contains the query results, where $|B| = \mathcal{O}(\gamma)$ throughout the execution.*
3. $\textsf{Filter-Decrypt}(B, A_{private})$*: Decrypts an encrypted buffer B, produced by F as above, using the private key and produces output B^*, a collection of the matching documents from S.*

Definition 6. *(Correctness of a Private Filter Generator)*
Let $F = \textsf{Filter-Gen}(D, Q_K, A_{public}, A_{private}, m, \gamma)$, where D is a dictionary, Q_K is a query for keywords K, $m, \gamma \in \mathbb{Z}^+$ and $(A_{public}, A_{private}) = \textsf{Key-Gen}(k)$. We say that a private filter generator is correct *if the following holds:*
Let F run on any document stream S, and set $B = F(S)$.
Let $B^* = \textsf{Filter-Decrypt}(B, A_{private})$. Then,

- *If $|\{M \in S \mid Q_K(M) = 1\}| \leq m$ then*

$$\Pr\Big[B^* = \{M \in S \mid Q_K(M) = 1\}\Big] > 1 - neg(\gamma)$$

- *If $|\{M \in S \mid Q_K(M) = 1\}| > m$ then*

$$\Pr\Big[(B^* \subset \{M \in S \mid Q_K(M) = 1\}) \vee (B^* = \perp)\Big] > 1 - neg(\gamma)$$

where $\perp$ is a special symbol denoting buffer overflow, and the probabilities are taken over all coin-tosses of F, $\textsf{Filter-Gen}$ *and of* $\textsf{Key-Gen}$.

Definition 7. *(Privacy) Fix a dictionary D. Consider the following game between an adversary A, and a challenger C. The game consists of the following steps:*

1. *C first runs* $\textsf{Key-Gen}(k)$ *to obtain $A_{public}, A_{private}$, and then sends A_{public} to A.*
2. *A chooses two queries for two sets of keywords, $Q_{0_{K_0}}, Q_{1_{K_1}}$, with $K_0, K_1 \subset D$ and sends them to C.*
3. *C chooses a random bit $b \in \{0, 1\}$ and executes* $\textsf{Filter-Gen}(D, Q_{b_{K_b}}, A_{public}, A_{private}, m, \gamma)$ *to create F_b, the filtering program for the query $Q_{b_{K_b}}$, and then sends F_b back to A.*

4. $A(F_b)$ can experiment with code of F_b in an arbitrary non-black-box way finally output $b' \in \{0,1\}$.

The adversary wins the game if $b = b'$ and loses otherwise. We define the adversary A's advantage in this game to be $\mathrm{Adv}_A(k) = \left|\Pr(b = b') - \frac{1}{2}\right|$ We say that a private filter generator is semantically secure *if for any adversary $A \in PPT$ we have that $\mathrm{Adv}_A(k)$ is a negligible function, where the probability is taken over coin-tosses of the challenger and the adversary.*

2.2 Combinatorial Lemmas

We require in our definitions that matching documents are saved with overwhelming probability in the buffer B (in terms of the size of B), while non-matching documents are not saved at all (at worst, with negligible probability). We accomplish this by the following method: If the document is of interest to us, we throw it at random γ times into the buffer. What we are able to guarantee is that if only one document lands in a certain location, and no other document lands in this location, we will be able to recover it. If there is a collision of one or more documents, we assume that all documents at this location are lost (and furthermore, we guarantee that we will detect such collisions with overwhelming probability). To amplify the probability that matching documents survive, we throw each γ times, and we make the total buffer size proportional to $2\gamma m$, where m is the upper bound on the number of documents we wish to save. Thus, we need to analyze the following combinatorial game, where each document corresponds to a ball of different color.

Color-survival game: Let $m, \gamma \in \mathbb{Z}^+$, and suppose we have m different colors, call them $\{color_i\}_{i=1}^{m}$, and γ balls of each color. We throw the γm balls uniformly at random into $2\gamma m$ bins, call them $\{bin_j\}_{j=1}^{2\gamma m}$. We say that a ball "survives" in bin_j, if no other ball (of any color) lands in bin_j. We say that $color_i$ "survives" if at least one ball of color $color_i$ survives. We say that the game *succeeds* if *all* m colors survive, otherwise we say that it *fails*.

Lemma 1. *The probability that the* color-survival game fails *is negligible in γ.*

Proof: See full version.

Another issue is how to distinguish valid documents in the buffer from collisions of two or more matching documents in the buffer. (In general it is unlikely that the sum of two messages in some language will look like another message in the same language, but we need to guarantee this fact.) This can also be accomplished by means of a simple probabilistic construction. We will append to each document k bits, where exactly $k/3$ randomly chosen bits from this string are set to 1. When reading the buffer results, we will consider a document to be good if exactly $k/3$ of the appended bits are 1's. If a buffer collision occurs between two matching documents, the buffer at this location will store the sum of the messages, and the sum of 2 or more of the k-bit strings. [3] We need to analyze the probability that after adding the k-bit strings, the resulting string *still* has exactly $k/3$ bits set to 1, and show that this probability is negligible in k. We will phrase the problem in terms of "balls in bins" as before: let $Bins = \{bin_j\}_{j=1}^{k}$ be distinguishable bins, and let $\mathcal{T}(Bins)$ denote the process of selecting $k/3$ bins uniformly at random from all $\binom{k}{k/3}$ choices, and putting one ball in each of these bins. For a fixed randomness, we can formalize $\mathcal{T}$ as a map $\mathcal{T} : \mathbb{Z}^k \to \mathbb{Z}^k$ such that for any $v = (v_1, ..., v_k) \in \mathbb{Z}^k$, $0 \le (\mathcal{T}(v)_j - v_j) \le 1$ for all $j \in \{1, ..., k\}$, and $\sum_{j=1}^{k}(\mathcal{T}(v)_j - v_j) = k/3$. Let $N(bin_j)$ be the number of balls in bin_j. Now, after independently repeating this experiment with the same bins, which were initially empty, we will be interested in the probability that for exactly $2k/3$ bins, the number of balls inside is 0 mod n, for $n > 1$. I.e., after applying $\mathcal{B}$ twice, what is $\Pr[\ |\{j \mid N(bin_j) \equiv 0 \bmod n\}| = 2k/3]$?

Lemma 2. *Let $Bins = \{bin_j\}_{j=1}^{k}$ be distinguishable bins, all of which are empty. Let $Bins = \mathcal{T}^2(Bins)$. Then for all $n > 1$,*

$$\Pr\Big[|\{j \in \{1, ..., k\} \mid N(bin_j) \equiv 0 \ mod\ n\}| = 2k/3\Big]$$

is negligible in k.

Proof See full version.

2.3 Organization of the Rest of this Paper

In what follows, we will give several constructions of private filter generators, beginning with our most efficient construction using a variant of the Paillier Cryptosystem [18],[9]. We also show a construction with reduced program size using the Cachin-Micali-Stadler PIR protocol [5], then we give a construction based on any group homomorphic semantically secure encryption scheme, and finally a construction based on the work of Boneh, Goh, and Nissim [2] that extends the query semantics to include a single "AND" operation without increasing the program size.

3 Paillier-Based Construction

Definition 8. *Let $(G_1, \cdot), (G_2, *)$ be groups. Let $\mathcal{E}$ be the probabilistic encryption algorithm and $\mathcal{D}$ be the decryption algorithm of an encryption scheme with plaintext set G_1 and ciphertext set G_2. The encryption scheme is said to be* group homomorphic *if the encryption map $\mathcal{E} : G_1 \to G_2$ has the following property:*

$$\forall\ a, b \in G_1,\ \mathcal{D}(\mathcal{E}(a \cdot b)) = \mathcal{D}(\mathcal{E}(a) * \mathcal{E}(b))$$

[3] If a document does not match, it will be encrypted as the 0 message, as will its appended string of k bits, so nothing will ever be marked as a collision with a non-matching document.

Note that since the encryption is probabilistic, we have to phrase the homomorphic property using $\mathcal{D}$, instead of simply saying that $\mathcal{E}$ is a homomorphism. Also, as standard notation when working with homomorphic encryption as just defined, we will use id_{G_1}, id_{G_2} to be the identity elements of G_1, G_2, respectively.

3.1 Private Filter Generator Construction

We now formally present the Key-Gen, Filter-Gen, and Buffer-Decrypt algorithms. The class $\mathcal{Q}$ of queries that can be executed is the class of all boolean expressions using only $\vee$. By doubling the program size, it is easy to handle an $\vee$ of both presence and absence of keywords. For simplicity of exposition, we describe how to detect collisions separately from the main algorithm.

Key-Gen(k) Execute the key generation algorithm for the Paillier cryptosystem to find an appropriate RSA number, n and its factorization $n = pq$. We will make one additional assumption on $n = pq$: we require that $|D| < \min\{p, q\}$. (We need to guarantee that any number $\leq |D|$ is a unit mod n^s.) Save n as A_{public}, and save the factorization as $A_{private}$.

Filter-Gen($D, Q_K, A_{public}, A_{private}, m, \gamma$) This algorithm outputs a search program F for the query $Q_K \in \mathcal{Q}$. So, $Q_K(M) = \bigvee_{w \in K}(w \in M)$.
We will use the Damgård-Jurik extension [9] to construct F as follows. Choose an integer $s > 0$ based upon the size of documents that you wish to store so that each document can be represented as a group element in $\mathbb{Z}_{n^s}$. Then F contains the following data:

– A buffer B consisting of $2\gamma m$ blocks with each the size of two elements of $\mathbb{Z}^*_{n^{s+1}}$ (so, we view each block of B as an ordered pair $(v_1, v_2) \in \mathbb{Z}^*_{n^{s+1}} \times \mathbb{Z}^*_{n^{s+1}}$). Furthermore, we will initialize every position to $(1, 1)$, two copies of the identity element.
– An array $\widehat{D} = \{\widehat{d_i}\}_{i=1}^{|D|}$ where each $\widehat{d_i} \in \mathbb{Z}^*_{n^{s+1}}$ such that $\widehat{d_i}$ is an encryption of $1 \in \mathbb{Z}_{n^s}$ if $d_i \in K$ and is an encryption of 0 otherwise. (Note: We of course use re-randomized encryptions of these values for each entry in the array.)

F then proceeds with the following steps upon receiving an input document M from the stream:

1. Construct a temporary collection $\widehat{M} = \{\widehat{d_i} \in \widehat{D} \mid d_i \in M\}$.
2. Compute
$$v = \prod_{\widehat{d_i} \in \widehat{M}} \widehat{d_i}$$
3. Compute v^M and multiply (v, v^M) into γ random locations in the buffer B, just as in our combinatorial game from section 2.2.

Note that the private key actually is not needed. The public key alone will suffice for the creation of F.

Buffer-Decrypt($B, A_{private}$) First, this algorithm simply decrypts B one block at a time using the decryption algorithm for the Paillier system. Each decrypted block will contain the 0 message (i.e., $(0, 0)$) or a non-zero message, $(w_1, w_2) \in \mathbb{Z}_{n^s} \times \mathbb{Z}_{n^s}$. Blocks with the 0 message are discarded. A non-zero message (w_1, w_2) will be of the form

(c, cM') if no collisions have occurred at this location, where c is the number of distinct keywords from K that appear in M'. So to recover M', simply compute w_2/w_1 and add this to the array B^*. Finally, output B^*.

In general, the filter generation and buffer decryption algorithms will make use of Lemma 2, having the filtering software append an extra r bits to each document, with exactly $r/3$ bits equal to 1, and then having the buffer decryption algorithm save documents to the output B^* if and only if exactly $r/3$ of the extra bits are 1. In any of our constructions, this can be accomplished by adding r extra blocks the size of the security parameter to an entry in the buffer to represent the bits. However, this is undesirable in our Paillier-based construction, as this would cause an increase (by a factor of $r/2$) to the size of the buffer.

Lemma 3. *With $\mathcal{O}(k)$ additional bits added to each block of B, we can detect all collisions of matching documents with probability $> 1 - neg(k)$.*

Proof. Since $log(|D|)$ is much smaller than the security parameter k, we can encode the bits from Lemma 2 using $\mathcal{O}(k)$ bits. For further details, see the full version of this paper.

3.2 Correctness

We need to show that if the number of matching documents is less than m, then

$$Pr\Big[B^* = \{M \in S \mid Q_K(M) = 1\}\Big] > 1 - neg(\gamma)$$

and otherwise, we have that B^* is a subset of the matching documents (or contains the overflow symbol, $\perp$). Provided that the buffer decryption algorithm can distinguish collisions in the buffer from valid documents (see above remark) this equates to showing that non-matching documents are saved with negligible probability in B and that matching documents are saved with overwhelming probability in B. These two facts are easy to show.

Theorem 1. *Assuming that the Paillier (and Damgård-Jurik) cryptosystems are semantically secure, then the private filter generator from the preceding construction is semantically secure according to Definition 7.*

Proof. Denote by $\mathcal{E}$ the encryption algorithm of the Paillier/Damgård-Jurik cryptosystem. Suppose that there exists an adversary A that can gain a non-negligible advantage ϵ in our semantic security game from Definition 7. Then A could be used to gain an advantage in breaking the semantic security of the Paillier encryption scheme as follows: Initiate the semantic security game for the Paillier encryption scheme with some challenger C. C will send us an integer n for the Paillier challenge. For messages m_0, m_1, we choose $m_0 = 0 \in \mathbb{Z}_{n^s}$ and choose $m_1 = 1 \in \mathbb{Z}_{n^s}$. After sending m_0, m_1 back to C, we will receive $e_b = \mathcal{E}(m_b)$, an encryption of one of these two values. Next we initiate the private filter generator semantic security game with A. A will give us two queries

Q_0, Q_1 in $\mathcal{Q}$ for some sets of keywords K_0, K_1, respectively. We use the public key n to compute an encryption of 0, call it $e_0 = \mathcal{E}(0)$. Now we pick a random bit q, and construct filtering software for Q_q as follows: we proceed as described above, constructing the array $\widehat{D}$ by using re-randomized encryptions, $\mathcal{E}(0)$ of 0 for all words in $D \setminus K_q$, and for the elements of K_q, we use $\mathcal{E}(0)e_b$, which are randomized encryptions of m_b. Now we give this program back to A, and A returns a guess q'. With probability $1/2$, e_b is an encryption of 0, and hence the program that we gave A does not search for anything at all, and in this event clearly A's guess is independent of q, and hence the probability that $q' = q$ is $1/2$. However, with probability $1/2$, $e_b = \mathcal{E}(1)$, hence the program we've sent A is filtering software that searches for Q_q, constructed exactly as in the Filter-Gen algorithm, and hence in this case with probability $1/2 + \epsilon$, A will guess q correctly, as our behavior here was indistinguishable from an actual challenger. We determine our guess b' as follows: if A guesses $q' = q$ correctly, then we will set $b' = 1$, and otherwise we will set $b' = 0$. Putting it all together, we can now compute the probability that our guess is correct: $\Pr(b' = b) = \frac{1}{2}\left(\frac{1}{2}\right) + \frac{1}{2}\left(\frac{1}{2} + \epsilon\right) = \frac{1}{2} + \frac{\epsilon}{2}$ and hence we have obtained a non-negligible advantage in the semantic security game for the Paillier system, a contradiction to our assumption. Therefore, our system is secure according to Definition 7.

4 Reducing Program Size Below Dictionary Size

In our other constructions, the program size is proportional to the size of the dictionary. By relaxing our definition slightly, we are able to provide a new construction using Cachin-Micali-Stadler PIR [5] which reduces the program size. Security of this system depends on the security of [5] which uses the *Φ-Hiding Assumption.*[4]

The basic idea is to have a standard dictionary D agreed upon ahead of time by all users, and then to replace the array $\widehat{M}$ in the filtering software with PIR queries that execute on a database consisting of the characteristic function of M with respect to D to determine if keywords are present or not. The return of the queries is then used to modify the buffer. This will reduce the size of the distributed filtering software. However, as mentioned above, we will need to relax our definition slightly and publish an upper bound U for $|K|$, the number of keywords used in a search.

4.1 Private Filter Generation

We now formally present the Key-Gen, Filter-Gen, and Buffer-Decrypt algorithms of our construction. The class $\mathcal{Q}$ of queries that can be executed by this protocol is again just the set of boolean expressions in only the operator $\vee$ over presence

[4] It is an interesting open question how to reduce the program size under other cryptographic assumptions.

or absence of keywords, as discussed above. Also, an important note: for this construction, it is necessary to know the set of keywords being used during key generation, and hence what we achieve here is only weak public key program obfuscation, as in Definition 2. For consistency of notation, we still present this as 3 algorithms, even though the key generation could be combined with the filter generation algorithm. For brevity, we omit the handling of collision detection, which is handled using Lemma 2.

Key-Gen(k, K, D) The CMS algorithms are run to generate PIR queries, $\{q_j\}$ for the keywords K, and the resulting factorizations of the corresponding composite numbers $\{m_j\}$ are saved as the key, $A_{private}$, while A_{public} is set to $\{m_j\}$.

Filter-Gen($D, Q_K, A_{public}, A_{private}, m, \gamma$) This algorithm constructs and outputs a private filter F for the query Q_K, using the PIR queries q_j that were generated in the Key-Gen(k, K, D) algorithm. It operates as follows.
F contains the following data:

- The array of CMS PIR queries, $\{q_j\}_{j=1}^{U}$ from the first algorithm, which are designed to retrieve a bit from a database having size equal to the number of words in the agreed upon dictionary, D. Only $|K|$ of these queries will be meaningful. For each $w \in K$, there will be a meaningful query that retrieves the bit at index corresponding to w's index in the dictionary. Let $\{p_{j,l}\}_{l=1}^{|D|}$ be the primes generated by the information in q_j, and let m_j be composite number part of q_j. The leftover $U - |K|$ queries are set to retrieve random bits.
- An array of buffers $\{B_j\}_{j=1}^{U}$, each indexed by blocks the size of elements of $\mathbb{Z}_{m_j}^*$, with every position initialized to the identity element.

The program then proceeds with the following steps upon receiving an input document M:

1. Construct the complement of the characteristic vector for the words of M relative to the dictionary D. I.e., create an array of bits $\bar{D} = \{\bar{d}_i\}$ of size $|D|$, such that $\bar{d}_i = 0 \Leftrightarrow d_i \in M$. We'll use this array as our database for the PIR protocols.

   Next, for each $j \in \{1, 2, ..., U\}$, do the following steps:
2. Execute query q_j on $\bar{D}$ and store the result in r_j.
3. Bitwise encrypt M, using r_j to encrypt a 1 and using the identity of $\mathbb{Z}_{m_j}^*$ to encrypt a 0.
4. Take the jth encryption from step 3 and position-wise multiply it into a random location in buffer B_j γ-times, as described in our color-survival game from section 2.

Buffer-Decrypt($B, A_{private}$) Simply decrypts each buffer B_j one block at a time by interpreting each group element with $p_{j,i}$th roots as a 0 and other elements as 1's, where i represents the index of the bit that is searched for by query q_j. All valid non-zero decryptions are stored in the output B^*.

4.2 Correctness of Private Filter

Since CMS PIR is not deterministic, it is possible that our queries will have the wrong answer at times. However, this probability is negligible in the security parameter. Again, as we've seen before, provided that the decryption algorithm can distinguish valid documents from collisions in buffer, correctness equates to storing non-matching documents in B with negligible probability and matching documents with overwhelming probability. These facts are easy to verify.

Theorem 2. *Assume the Φ-Assumption holds. Then the Private Filter Generator from the preceding construction is semantically secure according to Definition 2.*

Proof. (Sketch) If an adversary can distinguish any two keyword sets, then the adversary can also distinguish between two fixed keywords, by a standard hybrid argument. This is precisely what it means to violate the privacy definition of [5].

5 Eliminating the Probability of Error

In this section, we present ways to eliminate the probability of collisions in the buffer by using perfect hash functions. Recall the definition of perfect hash function. For a set $S \subset \{1, ..., m\}$, if a function $h : \{1, ..., m\} \to \{1, ..., n\}$ is such that $h|_S$ (the restriction of h to S) is injective, then h is called a *perfect hash function* for S. We will be concerned with families of such functions. We say that H is an (m, n, k)-*family of perfect hash functions* if $\forall S \subset \{1, ..., m\}$ with $|S| = k$, $\exists h \in H$ such that h is perfect for S.

We will apply these families in a very straightforward way. Namely, we define m to be the number of documents in the stream and k to be the number of documents we expect to save. Then, since there exist polynomial size (m, n, k)-families of perfect hash functions H, e.g., $|H| \leq \left\lceil \frac{\log\binom{m}{k}}{\log(n^k) - \log(n^k - k!\binom{n}{k})} \right\rceil$ [16], then our system could consist of $|H|$ buffers, each of size n documents, and our protocol would just write each (potential) encryption of a document to each of the $|H|$ buffers once, using the corresponding hash function from H to determine the index in the buffer. Then, no matter which of the $\binom{m}{k}$ documents were of interest, at least one of the functions in H would be injective on that set of indexes, and hence at least one of our buffers would be free of collisions.

6 Construction Based On Any Homomorphic Encryption

We provide here an abstract construction based upon an arbitrary homomorphic, semantically secure public key encryption scheme. The class of queries $\mathcal{Q}$ that are considered here is again, all boolean expressions in only the operation $\vee$, over presence or absence of keywords, as discussed above. This construction is similar to the Paillier-based construction, except that since we encrypt bitwise, we incur an extra multiplicative factor of the security parameter k in the buffer size. However, both the proof and the construction are somewhat simpler and can be based on any homomorphic encryption.

6.1 Construction of Abstract Private Filter Generator

Throughout this section, let $\mathcal{PKE} = \{\mathcal{KG}(k), \mathcal{E}(p), \mathcal{D}(c)\}$ be a public key encryption scheme. Here, $\mathcal{KG}, \mathcal{E}, \mathcal{D}$ are key generation, encryption, and decryption algorithms, respectively for any group homomorphic, semantically secure encryption scheme, satisfying Definition 8. We describe the Key-Gen, Filter-Gen, and Buffer-Decrypt algorithms. We will write the group operations of G_1 and G_2 multiplicatively. (As usual, G_1, G_2 come from a distribution of groups in some class depending on the security parameter, but to avoid confusion and unnecessary notation, we will always refer to them simply as G_1, G_2 where it is understood that they are actually sampled from some distribution based on k.)

Key-Gen(k) Execute $\mathcal{KG}(k)$ and save the private key as $A_{private}$, and save the public parameters of $\mathcal{PKE}$ as A_{public}.

Filter-Gen($D, Q_K, A_{public}, A_{private}, m, \gamma$) This algorithm constructs and outputs a filtering program F for Q_K, constructed as follows.
F contains the following data:

- A buffer $B(\gamma)$ of size $2\gamma m$, indexed by blocks the size of an element of G_2 times the document size, with every position initialized to id_{G_2}.
- Fix an element $g \in G_1$ with $g \neq id_{G_1}$. The program contains an array $\widehat{D} = \{\widehat{d_i}\}_{i=1}^{|D|}$ where each $\widehat{d_i} \in G_2$ such that $\widehat{d_i}$ is set to $\mathcal{E}(g) \in G_1$ if $d_i \in K$ and it is set to $\mathcal{E}(id_{G_1})$ otherwise. (Note: we are of course re-applying $\mathcal{E}$ to compute each encryption, and not re-using the same encryption with the same randomness over and over.)

F then proceeds with the following steps upon receiving an input document M:

1. Construct a temporary collection $\widehat{M} = \{\widehat{d_i} \in \widehat{D} \mid d_i \in M\}$.
2. Choose a random subset $S \subset \widehat{M}$ of size $\lceil |\widehat{M}|/2 \rceil$ and compute
$$v = \prod_{s \in S} s$$
3. Bitwise encrypt M using encryptions of id_{G_1} for 0's and using v to encrypt 1's to create a vector of G_2 elements.
4. Choose a random location in B, take the encryption of step 3, and position-wise multiply these two vectors storing the result back in B at the same location.
5. Repeat steps 2-4 $\left(\frac{c}{c-1}\right)\gamma$ times, where in general, c will be a constant approximately the size of G_1.

Buffer-Decrypt($B, A_{private}$) Decrypts B one block at a time using the decryption algorithm $\mathcal{D}$ to decrypt the elements of G_2, and then interpreting non-identity elements of G_1 as 1's and id_{G_1} as 0, storing the non-zero, valid messages in the output B^*.

7 Construction For a Single AND

7.1 Handling Several AND Operations by Increasing Program Size

We note that there are several simple (and unsatisfactory) modifications that can be made to our basic system to compute an AND. For example a query consisting of at most a c AND operations can be performed simply by changing the dictionary D to a dictionary D' containing all $|D|^c$ c-tuples of words in D, which of course comes at a polynomial blow-up[5] of program size.[6] So, only constant, or logarithmic size keyword sets can be used in order to keep the program size polynomial.

7.2 Executing a Single AND Without Increasing Program Size

Using the results of Boneh, Goh, and Nissim [2], we can extend the types of queries that can be privately executed to include queries involving a single AND of an OR of two sets of keywords without increasing the program size. This construction is very similar to the abstract construction, and hence several details that would be redundant will be omitted from this section. The authors of [2] build an additively homomorphic public key cryptosystem that is semantically secure under this subgroup decision problem. The plaintext set of the system is $\mathbb{Z}_{q_2}$, and the ciphertext set can be either $\mathbb{G}$ or $\mathbb{G}_1$ (which are both isomorphic to $\mathbb{Z}_n$). However, the decryption algorithm requires one to compute discrete logs. Since there are no known algorithms for efficiently computing discrete logs in general, this system can only be used to encrypt small messages. Using the bilinear map e, this system has the following homomorphic property. Let $F \in \mathbb{Z}_{q_2}[X_1, ..., X_u]$ be a multivariate polynomial *of total degree 2* and let $\{c_i\}_{i=1}^u$ be encryptions of $\{x_i\}_{i=1}^u$, $x_i \in \mathbb{Z}_{q_2}$. Then, one can compute an encryption c_F of the evaluation $F(x_1, ..., x_u)$ of F on the x_i with only the public key. This is done simply by using the bilinear map e in place of any multiplications in F, and then multiplying ciphertexts in the place of additions occurring in F. And once again, note that decryption is feasible only when the x_i are small, so one must restrict the message space to be a small subset of $\mathbb{Z}_{q_2}$. (In our application, we will always have $x_i \in \{0, 1\}$.) Using this cryptosystem in our abstract construction, we can easily extend the types of queries that can be performed.

7.3 Construction of Private Filter Generator

More precisely, we can now perform queries of the following form, where M is a document and $K_1, K_2 \subset D$ are sets of keywords:

$$(M \cap K_1 \neq \varnothing) \wedge (M \cap K_2 \neq \varnothing)$$

[5] Asymptotically, if we treat $|D|$ as a constant, the above observation allows a logarithmic number of AND operations with polynomial blow-up of program size. It is an interesting open problem to handle more than a logarithmic number of AND operations, keeping the program size polynomial.

[6] A naive suggestion that we received for an implementation of "AND" is to keep track of several buffers, one for each keyword or set of keywords, and then look for documents that appear in each buffer after the buffers are retrieved, however this will put many non-matching documents in the buffers, and hence is inappropriate for the streaming model. Furthermore, it really just amounts to searching for an OR and doing local processing to filter out the difference.

We describe the Key-Gen, Filter-Gen, and Buffer-Decrypt algorithms below.

Key-Gen(k) Execute the key generation algorithm of the BGN system to produce $A_{public} = (n, \mathbb{G}, \mathbb{G}_1, e, g, h)$ where g is a generator, $n = q_1 q_2$, and h is a random element of order q_1. The private key, $A_{private}$ is the factorization of n. We make the additional assumption that $|D| < q_2$.

Filter-Gen($D, Q_{K_1,K_2}, A_{public}, A_{private}, m, \gamma$) This algorithm constructs and outputs a private filter F for the query Q_{K_1,K_2}, constructed as follows, where this query searches for all documents M such that $(M \cap K_1 \neq \varnothing) \wedge (M \cap K_2 \neq \varnothing)$.
F contains the following data:

– A buffer $B(\gamma)$ of size $2\gamma m$, indexed by blocks the size of an element of $\mathbb{G}_1$ times the document size, with every position initialized to the identity element of $\mathbb{G}_1$.
– Two arrays $\widehat{D_l} = \{\widehat{d_i^l}\}_{i=1}^{|D|}$ where each $\widehat{d_i^l} \in \mathbb{G}$, such that $\widehat{d_i^l}$ is an encryption of $1 \in \mathbb{Z}_n$ if $d_i \in K_l$ and an encryption of 0 otherwise.

F then proceeds with the following steps upon receiving an input document M:

1. Construct temporary collections $\widehat{M_l} = \{\widehat{d_i^l} \in \widehat{D_l} \mid d_i \in M\}$.
2. For $l = 1, 2$, compute
$$v_l = \prod_{\widehat{d_i^l} \in \widehat{M_l}} \widehat{d_i^l}$$
and
$$v = e(v_1, v_2) \in \mathbb{G}_1$$
3. Bitwise encrypt M using encryptions of 0 in $\mathbb{G}_1$ for 0's and using v to encrypt 1's to create a vector of $\mathbb{G}_1$ elements.
4. Choose γ random locations in B, take the encryption of step 3, and position-wise multiply these two vectors storing the result back in B at the same location.

Buffer-Decrypt($B, A_{private}$) Decrypts B one block at a time using the decryption algorithm from the BGN system, interpreting non-identity elements of $\mathbb{Z}_{q_2}$ as 1's and 0 as 0, storing the non-zero, valid messages in the output B^*.[7]

Theorem 3. *Assuming that the subgroup decision problem of [2] is hard, then the Private Filter Generator from the preceding construction is semantically secure according to Definition 7.*

References

1. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of software obfuscation. In *Crypto 2001*, pages 1–18, 2001. LNCS 2139.
2. D. Boneh, E. Goh, K. Nissim. Evaluating 2-DNF Formulas on Ciphertexts. TCC 2005: 325-341

[7] See footnote 3.

3. D. Boneh, G. Crescenzo, R. Ostrovsky, G. Persiano. Public Key Encryption with Keyword Search. EUROCRYPT 2004: 506-522
4. Y. C. Chang. Single Database Private Information Retrieval with Logarithmic Communication. ACISP 2004
5. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *Advances in Cryptology – EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.
6. B. Chor, N. Gilboa, M. Naor Private Information Retrieval by Keywords in Technical Report TR CS0917, Department of Computer Science, Technion, 1998.
7. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proc. of the 36th Annu. IEEE Symp. on Foundations of Computer Science*, pages 41–51, 1995. Journal version: *J. of the ACM*, 45:965–981, 1998.
8. G. Di Crescenzo, T. Malkin, and R. Ostrovsky. Single-database private information retrieval implies oblivious transfer. In *Advances in Cryptology - EUROCRYPT 2000*, 2000.
9. I. Damgård, M. Jurik. A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System. In Public Key Cryptography (PKC 2001)
10. M. Freedman, Y. Ishai, B. Pinkas and O. Reingold. Keyword Search and Oblivious Pseudorandom Functions. To appear in 2nd Theory of Cryptography Conference (TCC '05) Cambridge, MA, Feb 2005.
11. S. Goldwasser and S. Micali. Probabilistic encryption. In J. Comp. Sys. Sci, 28(1):270–299, 1984.
12. K. Kurosawa, W. Ogata. Oblivious Keyword Search. Journal of Complexity, Volume 20 , Issue 2-3 April/June 2004 Special issue on coding and cryptography Pages: 356 - 371
13. E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proc. of the 38th Annu. IEEE Symp. on Foundations of Computer Science*, pages 364–373, 1997.
14. E. Kushilevitz and R. Ostrovsky. One-way Trapdoor Permutations are Sufficient for Non-Trivial Single-Database Computationally-Private Information Retrieval. In *Proc. of EUROCRYPT '00*, 2000.
15. H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. IACR ePrint Cryptology Archive 2004/063
16. K. Mehlhorn. On the Program Size of Perfect and Universal Hash Functions. In Proc. 23rd annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 170-175.
17. M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. *Proc. 31st STOC*, pp. 245–254, 1999.
18. P. Paillier. Public Key Cryptosystems based on Composite Degree Residue Classes. Advances in Cryptology - EUROCRYPT '99, LNCS volume 1592, pp. 223-238. Springer Verlag, 1999.
19. T. Sander, A. Young, M.Yung. Non-Interactive CryptoComputing For NC1 FOCS 1999: 554-567
20. J.P. Stern, A New and Efficient All or Nothing Disclosure of Secrets Protocol Asiacrypt 1998 Proceedings, Springer Verlag.