of not being function hiding) given that it involves only standard encryption and decryption operations.

3. PRELIMINARIES

In this section, we describe the tools we use to build our schemes, explain the adversary model, and establish notation.

IND-CCA2 Secure Encryption. IND-CCA2 encryption provides security against adaptive chosen-ciphertext attacks. It has the following useful non-malleability property: given an encryption of a message m, it is infeasible for a computationally bounded adversary to create an encryption of a message related to m. The security requirement is formally captured via an indistinguishability based security *game* which we briefly describe here. An adversary outputs two messages m_0 and m_1 and is given a ciphertext c – an encryption of either m_0 or m_1. Even with access to a decryption oracle which can be used to decrypt any ciphertext except c itself, adversary should not be able to tell which message c corresponds to. Very efficient IND-CCA2 encryption schemes are known in the random oracle model, for instance RSA-OAEP [27]. A detailed discussion of the relationship among different notions of security for public-key encryption can be found in [14].

Oblivious transfer. Oblivious transfer (OT) is one of the most widely studied fundamental primitives in cryptography. It is a two party protocol between a sender, who has two strings x_0 and x_1, and a chooser with choice bit b. While sender does not learn anything about the bit b (chooser's security) in the protocol, chooser only learns the value of x_b (sender's security).

In the common random string (CRS) model, Piekert et al. [54] give an efficient, one-round (first message from chooser to sender, next one from sender to chooser), UC-secure protocol for OT under the well-studied DDH assumption (as well as a variety of other hardness assumptions). We use this protocol, denoted by Π_{OT}, in our general construction in Section 5.2.

Garbled Circuits. The main component of our general construction is garbled circuits (GC), introduced by Yao in [60]. See [15] for a recent treatment of this tool. Several implementations of GC already exist [48, 36], and lately, much work has been going into making it more secure and efficient [46, 45, 38, 47, 37, 61].

Adversary model. The two most widely studied corruption models in literature are honest-but-curious and malicious[5]. Honest-but-curious parties follow their designated protocol diligently but try to learn the secrets of other parties from the interactions they have with them. On the other hand, malicious parties behave the way they like: not only they try to learn more information, but also execute any protocol of their choice in order to do so. In this work, our constructions provide security against honest-but-curious authorities and malicious clients.

Notation. We use κ to denote the security parameter. A function is negligible in κ (denoted $negl(\kappa)$) if it is smaller than the inverse of any polynomial, for all large enough values of κ. A probabilistic polynomial time algorithm, denoted in short by PPT, is an algorithm whose running time is bounded by some polynomial in κ on all inputs. This algorithm may use random coin tosses during its execution.

We use $\mathbb{Z}_N$ to denote the ring of integers modulo N (consisting of 0 and first $N-1$ positive integers with addition and multiplication defined modulo N). A vector of ℓ elements in this ring is denoted by $\vec{x} = (x_1, x_2, \ldots, x_\ell)$, where every $x_i \in \mathbb{Z}_N$. Encryption of a vector $\vec{x}$ means encryption of the concatenation of the elements of $\vec{x}$.

We use $[1, n]$ to denote the set $\{1, 2, \ldots, n\}$. For two strings a and b, $a \circ b$ denotes their concatenation and $a \oplus b$ denotes their bit-wise XOR. We denote the inner-product of two vectors $\vec{x}$ and $\vec{y}$ as $\langle \vec{x}, \vec{y} \rangle$

On using honest-but-curious third party. We assume the existence of an honest-but-curious third party (central authority) that is trusted not to collude with the party computing a function (client). In general, the whole public key infrastructure (PKI) depends upon trusted third parties called certificate authorities (CAs). PKI is widely used for securing communication on Internet. CAs are trusted not to collude with the adversary, but still be secure enough against attacks. This type of non-collusion assumption is widely used in literature. Indeed, primitives like Identity-Based Encryption, Attribute-Based Encryption, etc., all involve a central authority trusted to not collude with other parties in the system. Nikolaenko et al. [51, 50] use similar non-collusion assumption where they trust the third party (garbled circuit generator) not to collude with the garbled circuit evaluator. Protocols for outsourcing multi-party computation [23, 22, 42, 43] use similar non-collusion assumptions to outsource evaluation of the garbled circuits.

4. DEFINING C-FE

In this section, we formally define controlled functional encryption (C-FE) and the security models under which we study it.

Our definition of C-FE differs from the definition of FE [19] in a crucial way. In FE, once a client receives a key for a function f from the authority, it can use the key to decrypt any number of ciphertexts it wants. However, in our *controlled* setting, every time a client wants to decrypt a ciphertext CT_x with f, a key is requested from the authority. The authority generates a *one-time* key which could only be used to compute $f(x)$. In order to decrypt a different ciphertext, a new key request must be submitted. Consequently, we have an additional algorithm KeyReq in our definition of C-FE below. Further, the authority will need to extract the policy parameter attached to CT_x from the key request, before it can decide whether to honor the request; for this, an algorithm $\mathsf{Extract}$ is used.

Syntax. We start by defining the syntax and (perfect) correctness requirement for the six algorithms that form a C-FE scheme. The role of these algorithms is illustrated in Figure 1.

DEFINITION 4.1. *A controlled functional encryption (C-FE) scheme for a function family F_κ defined over $(\mathcal{F}_\kappa, \mathcal{X}_\kappa, \Lambda_\kappa)$ consists of six PPT algorithms $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyReq}, \mathsf{Extract}, \mathsf{KeyGen}, \mathsf{Dec})$ satisfying the following correctness condition for all $\kappa \in \mathbb{N}$, $x \in \mathcal{X}_\kappa$, $\lambda \in \Lambda_\kappa$ and $f \in \mathcal{F}_\kappa$. Consider the following experiment:*

$$
\begin{aligned}
(\mathsf{MPK}, \mathsf{MSK}) &\leftarrow \mathsf{Setup}(1^\kappa) \\
\mathsf{CT} &\leftarrow \mathsf{Enc}(x, \lambda, \mathsf{MPK}) \\
(\rho, \zeta) &\leftarrow \mathsf{KeyReq}(\mathsf{CT}, f) \\
(\lambda', \xi) &\leftarrow \mathsf{Extract}(\rho, \mathsf{MSK}) \\
\tau &\leftarrow \mathsf{KeyGen}(\xi) \\
z &\leftarrow \mathsf{Dec}(\zeta, \tau)
\end{aligned}
$$

Then we require $\lambda' = \lambda$ (the authority receives the policy parameter correctly) and $z = F_\kappa(f, x)$ (the decryption yields the correct output) with probability 1.

Note that instead of writing $f(x)$, we have denoted it as $F_\kappa(f, x)$, where F_κ could be considered a function family, and f specifies which function in the family should be applied to x.

[5] Honest-but-curious adversaries are also referred to as semi-honest or passive. Malicious adversaries are sometimes called active.

The above syntax does not explicitly accommodate a function-revealing or pattern-revealing C-FE scheme. In these variants, the output of Extract will contain not just λ, but also f and/or a unique identifier that can identify CT. To accommodate tweaks, we modify the definition of KeyGen to take a tweak $w \in \mathcal{W}_\kappa$ as an additional input, and change the final correctness requirement to $z = F_\kappa(f, x, w)$.

Security Definition. To cleanly capture the security guarantees of C-FE, we use an *ideal functionality*, in the spirit of Universally Composable (UC) security [21]. In UC security, ideal functionality is simply an ideal trusted party that captures the security guarantees: the various parties in the system (data owners, clients, authority) can learn only as much information as they can learn when interacting with this trusted party, and can influence the system only as much as the trusted party lets them. Our security definition is also along the lines of UC security, except that we consider an adversarial model in which the authority can only be corrupted passively (honest-but-curious) and only by itself (no collusion); while clients can be actively corrupted and may collude with each other.

Ideal World. We define the C-FE ideal functionality $\mathcal{F}$ that interacts with several "data owners," several "clients," and one "authority" as follows. (We omit routine details like initializing a session and how parties can join a session.) $\mathcal{F}$ can come in four modes of security, depending on whether it is function-hiding or function-revealing, and whether it is pattern-hiding or pattern-revealing.

1. A data-owner can upload a pair (x, λ) to $\mathcal{F}$. Then $\mathcal{F}$ picks a handle h (κ-bit random string) and sends it to all the clients. Internally, $\mathcal{F}$ records (h, x, λ) in a table.

2. A client can send an evaluation request (f, h) to $\mathcal{F}$. Then $\mathcal{F}$ proceeds as follows:

   (a) First, depending on its mode of security, $\mathcal{F}$ sends one of the following to the authority.

   - (f, h, λ), if function and pattern-revealing.
   - (h, λ), if function-hiding and pattern-revealing.
   - (f, λ), if function-revealing and pattern-hiding.
   - λ, if function and pattern-hiding.

   (b) Then it awaits for a command from the authority, either denying or allowing evaluation. In the former case, it sends a special symbol $\perp$ to the client. In the latter case, it receives a tweak w from the authority and sends $F(f, x, w)$ to the client.

Real World. In the "real world" execution, interaction with $\mathcal{F}$ is replaced by calls to a C-FE scheme, as follows. To initialize the system, the authority runs Setup first and publishes MPK (for the data owners). Instead of uploading (x, λ) to $\mathcal{F}$, a data owner would run Enc and privately communicate the resulting ciphertext CT to all the clients (in an implementation, an access controlled bulletin-board could be used to do this). Instead of sending an evaluation request to $\mathcal{F}$, a client would run KeyReq and send its output to the authority; the authority, instead of receiving λ (and possibly f and/or a handle for CT), uses Extract to obtain it. Then it can use an external decision process to decide whether or not to honor the key-request, and if it is to be honored, what the tweak value w should be. Then, instead of sending w to $\mathcal{F}$, the authority runs KeyGen to obtain a key that it sends to the client. Instead of receiving the output from $\mathcal{F}$, the client would compute it using Dec.

DEFINITION 4.2. *A* C-FE *scheme is simulation secure if for every adversary* Adv *in the real world who actively corrupts a subset of clients or only passively corrupts the authority alone, there is a simulator* Sim *in the ideal world execution, which also corrupts only the same set of parties, and produces an output identically distributed to* Adv*'s output in the real world.*

A more precise definition refers to "environments" in which the execution — real or ideal — takes place; for more details, we refer the reader to [21] and subsequent formulations of UC security.

5. CONSTRUCTIONS

In this section, we describe two C-FE schemes secure under the definitions discussed above. The first one is a simple and extremely efficient scheme for the inner-product functionality $\mathbf{F}_{\mathsf{IP}}$. It provides practical solutions to common problems like weighted average, hamming distance, etc. The second one is a general construction for any polynomial-time computable functionality. Though not as efficient as the first one, this construction is still practical for many problems of interest (as demonstrated by our experiments) and provides more security.

5.1 Inner Product

We first describe an efficient and simple way to compute inner-product between two vectors in our controlled functional encryption setting. Our scheme is secure against malicious clients and honest-but-curious authorities in the non-function hiding security model. Once again, note that we are able to compute the actual value of inner product, and not just whether it is zero or not.

Let $\mathbf{F}_{\mathsf{IP}} = \{f_{\vec{v}} \mid \vec{v} \in \mathbb{Z}_N^\ell\}$ denote the inner-product function family, where $f_{\vec{v}}(\vec{x}) = \langle \vec{v}, \vec{x} \rangle = \sum_{i=1}^{\ell} v_i \cdot x_i \mod N$ for all $\vec{x} \in \mathbb{Z}_N^\ell$. Note that to specify a function in this family, one only needs to provide the index $\vec{v}$. (For simplicity we have omitted the security parameter κ in the description.)

Overview. We first provide an overview of the construction. To encode an input vector $\vec{x}$, the data owner chooses a random vector $\vec{r}$ of the same dimension as $\vec{x}$ over $\mathbb{Z}_N$. It then outputs $(\vec{y}, \sigma)$, where $\sigma = Enc_{PK}(\vec{r})$ and Enc is a CCA2 secure PKE, and $\vec{y} = \vec{x} + \vec{r}$ (all operations over $\mathbb{Z}_N$). Note that $\vec{y}$ and $\vec{r}$ form an additive secret sharing of $\vec{x}$, and neither by itself contains any information about $\vec{x}$. The key-request computing $\langle \vec{x}, \vec{v} \rangle$ consists of $(\sigma, \vec{v})$. The authority will decrypt σ to obtain $\vec{r}$ and returns $\langle \vec{v}, \vec{r} \rangle$ to the client. The client locally computes $\langle \vec{v}, \vec{y} \rangle$ and adds it to the negative of the value obtained from the authority, to obtain

$$\langle \vec{v}, \vec{y} \rangle - \langle \vec{v}, \vec{r} \rangle = \langle \vec{v}, \vec{x} \rangle.$$

Even if the client sends the same σ to the authority several times, it is easy to show that the client's view can be simulated perfectly in an ideal world, where the client only obtains $\langle \vec{v}, \vec{x} \rangle$ for the values of $\vec{v}$ it sent to the authority. As in the general construction, using CCA2 secure encryption ensures that a malicious client obtains no advantage by sending a σ it did not obtain from a data owner.

Note that the authority's computation here involves a decryption, and an inner product computation, and the client's computation involves merely an inner product computation. Further, this scheme can exploit the sparsity of $\vec{v}$, i.e. the client's computation is proportional to the number of non-zero elements in $\vec{v}$. In fact, even if all vector elements are non-zero, our performance evaluations show that this construction is extremely fast for genomic-scale inputs.

Construction. A (function and pattern revealing) C-FE scheme Π_{IP} for the function family $\mathbf{F}_{\mathsf{IP}}$ is presented in Figure 2 and a proof of security is described below. Correctness of Π_{IP} follows easily from construction and the linearity of dot product:

$$\langle \vec{v}, \vec{y} \rangle - \tau = \langle \vec{v}, \vec{x} + \vec{r} \rangle - \langle \vec{v}, \vec{r} \rangle = \langle \vec{v}, \vec{x} \rangle.$$

Note that we can allow the authority to add noise to the outcome using a tweak: $\mathsf{KeyGen}_{\mathsf{IP}}$ will take the noise w as an additional input and set $\tau = \langle \vec{v}, \vec{r} \rangle - w$, so that the outcome obtained by the client is $\langle \vec{v}, \vec{x} \rangle + w$.

Protocol Π_{IP}

- $\mathsf{Setup}_{\mathsf{IP}}(1^\kappa)$: Run $\mathsf{Setup}_{\mathsf{CCA2}}(1^\kappa)$ to obtain $(\mathsf{PK}, \mathsf{SK})$. Set MPK and MSK to be PK and SK respectively.
- $\mathsf{Enc}_{\mathsf{IP}}(\vec{x}, \lambda, \mathsf{MPK})$: Choose ℓ numbers $r_1, r_2, \ldots, r_\ell$ uniformly at random from $\mathbb{Z}_N$. Let $\vec{r} = (r_1, r_2, \ldots, r_\ell)$. Output the ciphertext $\mathsf{CT} = (\vec{x} + \vec{r}, \mathsf{Enc}_{\mathsf{CCA2}}(\vec{r}, \lambda, \mathsf{MPK}))$.
- $\mathsf{KeyReq}_{\mathsf{IP}}(\mathsf{CT}, \vec{v})$: Let $\mathsf{CT} = (\vec{y}, \sigma)$. Output $\rho = (\sigma, \vec{v})$ and $\zeta = (\vec{v}, \vec{y})$.
- $\mathsf{Extract}_{\mathsf{IP}}(\rho, \mathsf{MSK})$: Let $\rho = (\sigma, \vec{v})$. Run $\mathsf{Dec}_{\mathsf{CCA2}}(\sigma, \mathsf{MSK})$ to obtain $(\vec{r}, \lambda)$. Output $((\lambda, \sigma), \xi)$ where σ is used to reveal the pattern, and $\xi = (\vec{v}, \vec{r})$.
- $\mathsf{KeyGen}_{\mathsf{IP}}(\xi)$: Let $\xi = (\vec{v}, \vec{r})$. Output $\tau = \langle \vec{v}, \vec{r} \rangle$.
- $\mathsf{Dec}_{\mathsf{IP}}(\zeta, \tau)$: Let $\zeta = (\vec{v}, \vec{y})$. Output $\langle \vec{v}, \vec{y} \rangle - \tau$.

Figure 2: Superfast Construction for computing actual inner product.

Proof of Security. A simple simulator, combined with the CCA2 security of the encryption scheme can be used to prove the security. The interesting case is when a client is corrupt. To translate the ideal world view to the real world view of the client, we consider the following simulator.

First, the simulator picks a pair of keys $(\mathsf{MPK}, \mathsf{MSK})$ to simulate the setup. Then, when it receives a handle h, it creates a simulated ciphertext $\mathsf{CT}' = (\vec{y_h}, c_h)$, where $\vec{y_h}$ is a random vector (which is identically distributed as $\vec{x} + \vec{r}$ in the real ciphertext), and $c_h \leftarrow \mathsf{Enc}_{\mathsf{CCA2}}(0^{|(x,\lambda)|}, \mathsf{MPK})$ is a ciphertext encrypting a string of zeros (which will be indistinguishable from the encryption of (x, λ)). Later, if the client sends out a key-request of the form $(\sigma, \vec{v})$, the simulator checks if $\sigma = c_h$ for some handle h. There are two cases:

1. $\sigma = c_h$ for some h: Then the simulator forwards an evaluation request $(h, \vec{v})$ to the ideal functionality, which will return $z = \langle \vec{v}, \vec{x} \rangle$. The simulator creates a simulated value $\tau' = \langle \vec{v}, \vec{y_h} \rangle + z$. In this case, τ' is *identically distributed* as the value τ the client receives in the real execution.

2. There is no h s.t. $\sigma = c_h$: In this case the simulator acts like the authority. i.e., It decrypts σ and (if the decryption is valid) obtains some vector $\vec{r}$; then it returns $\tau' = \langle \vec{v}, \vec{r} \rangle$.

In the second case above the ciphertext σ sent by the client to the authority does not correspond to any (simulated) ciphertext it received (from any simulated data owner). However, the client could have created σ in an arbitrary fashion, without necessarily encrypting a value r that it knows. To argue that the simulation is indistinguishable from the real execution, we need to argue that the dummy ciphertexts c_h created by the simulator remain indistinguishable from real ciphertexts, even though the simulator carries out decryptions of arbitrary ciphertexts σ that the adversary presents (other than the dummy ciphertexts themselves). This is possible, thanks to the CCA2 security of the encryption scheme: a distinguisher between the real execution and the simulation can be turned into an adversary that distinguishes the encryptions of real messages and dummy messages, with the help of a decryption oracle (to which it never sends challenge ciphertexts).

5.2 General construction

In this section, we construct a controlled FE scheme Π for any polynomial-time computable family of functions $\mathcal{F}$, which take inputs from the domain $\mathcal{X}$. Without loss of generality, we assume that an element $f \in \mathcal{F}$ can be represented by ℓ bits, and an element $x \in \mathcal{X}$ can be represented by k bits, where both ℓ and k are some polynomial in κ.

Overview. We start by sketching an intuitive construction, which is neither secure nor efficient enough, and then describe how to fix these issues. For simplicity, we start with a function revealing, pattern revealing construction, with no tweaks, for general function evaluation. That is, we are given an arbitrary (efficiently computable) function F so that the client should be able to compute $F(f, x)$, if the authority, after seeing λ (and pattern information), allows the client to do so, where $(x; \lambda)$ is a piece of data and associated policy parameter from a data owner, and f is a function specification from the client.

First, the authority runs a Setup algorithm: it picks an encryption-decryption key pair $(\mathsf{MPK}, \mathsf{MSK})$ for a public-key encryption scheme, and publishes the public key. The encryption algorithm used by the data owner takes (x, λ) and creates two ciphertexts (α, σ), obtained by encrypting (x, λ) respectively, under PK. The client will receive (α, σ) and when it wants to evaluate $f(x)$, it sends a key-request to the authority consisting of (f, σ). The authority can recover λ by decrypting σ; it can also update the pattern information, if necessary, by comparing λ with previous key requests it received. Then, if it decides to honor the key-request, it engages in a one-round 2-party secure computation (using garbled circuits and a one-round OT protocol, for instance) to compute the following function F'. F' takes α as input from the client and (f, SK) as input from the authority, and outputs $F(Dec_{SK}(\alpha), f)$ to the client. Note that $Dec_{SK}(\alpha) = x$. This would fit the syntax of C-FE, if the key-request includes the first message from the client to the authority in the secure computation protocol. As for security, since a secure computation protocol is used, the client should learn nothing other than $F(f, x)$.

There are two major problems with this solution:

1. Firstly, it is not secure against malicious clients. In particular, suppose the client feeds not α, but a related ciphertext α' to the secure computation protocol. Then the client can potentially learn $F(x', f)$ where $x' = Dec_{SK}(\alpha')$ is related to x.

2. Secondly, the above solution has a serious drawback in terms of efficiency. Often F is a very simple function (like inner product or Hamming distance), and can be very efficiently implemented using a 2-party secure computation protocol. However, the secure computation protocol used above is not for evaluating F, but for evaluating F'. Note that F' involves a (public-key) decryption operation. This decryption applies to a ciphertext of the entire input x. This makes the scheme vastly inefficient and often impractical.

The first problem is easy to fix. To thwart all such malleability attacks we can use a CCA2 secure public-key encryption scheme. (One should also bind α and σ together using a random nonce, so that the client cannot replace the policy of one data owner with that of another; our final solution will not have this structure, and hence will not need the use of a random nonce.)

To address the second problem, we need to ensure that the secure computation is for F itself, and not a function like F'. To achieve this, we take a closer look at how a garbled circuit based 2-party computation protocol proceeds. The client can evaluate a garbled circuit only for an input for which it holds the requisite "labels": each bit position of input has two labels associated with it, corresponding to the values 0 and 1, that are picked at random by the garbled circuit generator. To let a client evaluate the circuit on a k-bit input x (belonging to the generator) that the client does not

know, the generator sends the k labels corresponding to x, without revealing whether each label corresponds to value 0 or 1.

In our case, unfortunately, the garbled circuit is generated by the authority who also does not know x, and it will not be able to send just the relevant labels. However, it can take the help of the data owner (who is offline), as follows.

Recall that the data owner encodes x as (α, σ) where α is given to the client and σ to the authority. σ will contain a set of $2k$ keys for symmetric-key encryption (SKE), encrypted under the authority's public-key. The authority will encrypt *all the* $2k$ *labels* (2 per bit position of x) under these keys. Further, the message inside σ will also specify a random order for the 2 encrypted labels for each bit position. The authority will send all the encrypted labels to the client in this order. Note that σ contains no information about x itself. Now, the client needs to recover only the k labels corresponding to x from these $2k$ encrypted labels. For this, α will include k SKE keys (in the clear), one out of the two keys for each bit position, corresponding to the bit value of x at that position. α will also indicate, for each bit position, whether the given key corresponds to the first or the second one in the pair of encrypted labels that will be sent to the client by the authority. This allows the client to decrypt exactly the k labels corresponding to the bit values of x. Note that the client's view too does not contain any information about x, since which one in a pair of encrypted labels corresponds to the bit value 0 and which one to 1 is not known to the client.

We have another construction which is actually a slight optimization of the above scheme, that avoids the $2k$ encryptions by the authority and the k decryptions by the client, by having σ and α specify the labels themselves. The authority will pick all other labels for the garbled circuit freshly, but the $2k$ labels corresponding to the input wires for x remain the same.

Note that when f is known to the authority, the only labels that the authority cannot send directly to the client are those for x. In this case, the entire scheme is based only on public and symmetric-key encryption schemes. However, if we require function hiding, the authority will need to transfer the correct labels corresponding to f via oblivious-transfer.

The variants can be easily accommodated in this construction. Firstly, the message in the ciphertext σ can also contain the policy parameters λ associated with a piece of data x. Tweaks are simply additional inputs to F that the authority can hard-wire into the circuit, while creating the garbled circuit. To allow pattern hiding, we replace CCA2 secure encryption with Rerandomizable RCCA secure encryption.

Construction. For the sake of simplicity, we ignore the policy parameter λ in the construction; this lets us merge the algorithms $\mathsf{Extract}$ and KeyGen (ignoring the need to extract λ). Let $(\mathsf{Setup}_{\mathsf{SKE}}, \mathsf{Enc}_{\mathsf{SKE}}, \mathsf{Dec}_{\mathsf{SKE}})$ be a symmetric key encryption scheme which generates keys of length κ and encrypts κ-length messages. Also, let Π_{OT} be a one-round oblivious transfer protocol where the first message is sent from chooser to sender and the second message from sender to chooser (for more details, see the paragraph 'oblivious transfer' in Section 3). Using these tools along with others, we present a formal construction of Π in Figure 3. The two messages (from the client to the authority, and back) in the OT protocol are combined with key-request and key-generation algorithms, so that the syntax of C-FE is respected. A proof of security is given below. We also present an alternate construction, which is slightly more efficient, in Appendix A.

Proof of security. We provide a sketch of the proof of security of Π. Once again, the interesting case is when a client is malicious. Let $\mathsf{Sim}_{\mathsf{GC}}$ be the simulator $\mathcal{S}$ of the projective prv.sim secure garbling scheme[6] with circuit being the side information i.e. $\mathsf{PrvSim}_{\mathcal{G},\phi,\mathcal{S}}$ where $\phi(f) = f$, as described in [15]. We construct a simulator Sim which uses $\mathsf{Sim}_{\mathsf{GC}}$ to simulate the view of a corrupt client Adv in the ideal world.

Sim runs Adv internally as a black-box. He first executes $\mathsf{Setup}(1^\kappa)$ to obtain $(\mathsf{MPK}, \mathsf{MSK})$, and sends MPK to Adv. When he receives a handle h from the ideal functionality, he chooses $(r_1^0, r_1^1), \ldots, (r_k^0, r_k^1)$ and v at random. Let $\hat{r} = r_1^0 \circ r_1^1 \circ r_2^0 \circ r_2^1 \ldots \circ r_k^0 \circ r_k^1$, and $\alpha = (r_1^0 \circ r_2^0 \ldots r_k^0, 0^{|x|})$. Sim provides (α, σ) as ciphertext to Adv, where $\sigma = \mathsf{Enc}_{\mathsf{CCA2}}(\hat{r} \circ v, \mathsf{MPK}))$.

When Adv initiates a key request $\rho = (\sigma', M)$, Sim first extracts an f from M (note that since Π_{OT} is a UC-secure protocol, this can be done). He then checks whether $\sigma' = \sigma$ for some h or not. If $\sigma' \neq \sigma$ for any h, Sim simply runs KeyGen with $(\sigma', M, \mathsf{MSK})$, and returns the output to Adv. In case there is equality for some h_x, he sends f and h_x to Eval, and obtains $F(f, x)$. He now invokes $\mathsf{Sim}_{\mathsf{GC}}$ with inputs $(f, F(f, x))$ to obtain a fake garbled circuit $\hat{C}_{\mathsf{Fake}}$. Having obtained the circuit, Sim runs the rest of KeyGen with $(r_1^0, r_1^1), \ldots, (r_k^0, r_k^1)$ and v (decrypted value of σ) and returns $(\hat{C}_{\mathsf{Fake}}, (c_1^0, c_1^1, \ldots, c_k^0, c_k^1), M')$ to Adv. This completes the description of Sim.

Protocol Π

- $\mathsf{Setup}(1^\kappa)$: Run $\mathsf{Setup}_{\mathsf{CCA2}}(1^\kappa)$ to obtain $(\mathsf{PK}, \mathsf{SK})$. Set MPK and MSK to be PK and SK respectively.
- $\mathsf{Enc}(x, \mathsf{MPK})$: For $i \in [1, k]$ and $b \in \{0, 1\}$, run $\mathsf{Setup}_{\mathsf{SKE}}(1^\kappa)$ to obtain a key r_i^b. Let $\hat{r} = r_1^0 \circ r_1^1 \circ r_2^0 \circ r_2^1 \ldots \circ r_k^0 \circ r_k^1$. Choose a uniformly random k-bit string v, and let $u = x \oplus v$. Now, let r_u denote $r_1^{u_1} \circ r_2^{u_2} \ldots r_k^{u_k}$, where u_i is the ith bit of u. Output the ciphertext $\mathsf{CT}_x = ((r_u, u), \mathsf{Enc}_{\mathsf{CCA2}}(\hat{r} \circ v, \mathsf{MPK}))$.
- $\mathsf{KeyReq}(\mathsf{CT}, f)$: Let $\mathsf{CT} = (\alpha, \sigma)$ and $f = (f_1, f_2, \ldots, f_\ell)$. For $j \in [1, \ell]$, run the first step of the oblivious transfer protocol Π_{OT} with chooser's input being f_j. Let M_j be the first message output by this protocol and R_j be the coin tosses used. Now, let M denote $(M_1, \ldots, M_\ell)$ and R denote $(R_1, \ldots, R_\ell)$. Output $\rho = (\sigma, M)$ and $\zeta = (\alpha, f, M, R)$.
- $\mathsf{KeyGen}(\rho = (\sigma, M), \mathsf{MSK})$: Run $\mathsf{Dec}_{\mathsf{CCA2}}(\sigma, \mathsf{MSK})$ to obtain $(r_1^0, r_1^1), \ldots, (r_k^0, r_k^1)$ and v. Consider the circuit C which takes as input a k-bit string z and an ℓ-bit function g, and computes $F(g, z)$. Construct a projective $\mathsf{PrvSim}_{\mathcal{G},\phi(f)=f,\mathcal{S}}$ garbled version of this circuit as described in [15] and call it $\hat{C}$, choosing keys at random for each and every wire of the circuit, including input wires. Let the key pairs corresponding to the k-bit string z be $(t_1^0, t_1^1), \ldots, (t_k^0, t_k^1)$. For $i \in [1, k]$ and $b \in \{0, 1\}$, run $\mathsf{Enc}_{\mathsf{SKE}}(t_i^b, r_i^{b \oplus v_i})$ to obtain a ciphertext $c_i^{b \oplus v_i}$, where v_i is the ith bit of v.
  Let the key pairs corresponding to the ℓ-bit function input g be $(s_1^0, s_1^1), \ldots, (s_\ell^0, s_\ell^1)$. Parse M as $(M_1, \ldots, M_\ell)$. For $j \in [1, \ell]$, run Π_{OT} with sender's input being (s_j^0, s_j^1) and message received from chooser being M_j. Let M_j' be the second message output by this protocol. Let M' denote $(M_1', \ldots, M_\ell')$. Output $\tau = (\hat{C}, (c_1^0, c_1^1, \ldots, c_k^0, c_k^1), M')$.
- $\mathsf{Dec}(\zeta, \tau)$: Parse τ as $(\hat{C}, (c_1^0, c_1^1, \ldots, c_k^0, c_k^1), (M_1', \ldots, M_\ell'))$ and ζ as $((r_u, u), f, (M_1, \ldots, M_\ell), (R_1, \ldots, R_\ell))$. For $j \in [1, \ell]$, run Π_{OT} with chooser's input f_j, coin tosses R_j, first round message M_j, and second round message M_j', to obtain the key for the jth bit of g. Parse r_u as $r_{u,1} \ldots r_{u,k}$. To find the key for the ith bit of z, run $\mathsf{Dec}_{\mathsf{SKE}}(c_i^{u_i}, r_{u,i})$ to obtain $t_i^{x_i}$. Now, evaluate $\hat{C}$ to obtain the value of $f(x)$.

Figure 3: General C-FE Construction

[6] The simulation based garbling schemes (prv.sim) and indistinguisbility based garbling scheme (prv.ind) schemes of [15] are equivalent in our setting.

Note that in the ideal world, when Sim receives a key request with σ', he generates a fake garbled circuit (GC) as opposed to a real circuit (if Sim creates a real GC, Adv would evaluate it to recover $f(0^{|x|})$, and distinguish the two worlds). However, a fake GC has the property that no matter what combination of keys a party uses for the input wires of the bits of x, the circuit always evaluates to $F(f, x)$. Therefore, even if Adv uses keys corresponding to $0^{|x|}$ to evaluate $\hat{C}_{\mathsf{Fake}}$, it will still recover $F(f, x)$. Hence, it cannot distinguish between a fake and a real GC.

Correctness follows easily from construction.

6. IMPLEMENTATION AND EVALUATION

We implemented our C-FE constructions: general as well as Superfast construction. Experiments were conducted on synthetic data, however, size of the data was inspired by the applications discussed in Section 7. We evaluated our general construction on a powerful machine but used a laptop to evaluate our Superfast inner-product construction.

6.1 Superfast Inner-Product Construction

Our Superfast inner-product construction is implemented in Java. We use RSA-OAEP [16] (RSA-Optimal Asymmetric Encryption Padding) implementation of the Java Cryptography Extension (JCE) and evaluated the construction with 1024-bit, 2048-bit and 4096-bit keys.

Experiments. Our Superfast inner-product construction is very light, so we used a laptop – with Intel Core i7 3615QM processor, 8GB memory and Mac OS X 10.9 – for the experiments.

Vector sizes. We used data vector of size 4,000,000[7] integers (4-byte) and varied the function vector size according to the different applications discussed in Section 7. We also used data vectors of size 40,000,000[8] integers for the evaluation.

Encryption in Superfast scheme has two parts: additive secret sharing of plaintext's field elements and public key-encryption of one share of each plaintext element. We concatenate several vector elements to be encrypted in one block to avoid blow up in the ciphertext size. Ideally, size of ciphertext in our scheme should be double the size of plaintext due to additive secret sharing. As, we use RSA-OAEP, padding makes size of the ciphertext little bit more than double. Per block padding size is same for different key sizes, but, as larger keys have larger block sizes, the ciphertext size decreases as the key size increases, as shown in Table 1.

Plaintext size (4-byte alphabet)	4,000,000			40,000,000
Plaintext size(MB)	15.26			152.59
Key size (bits)	1024	2048	4096	1024
Ciphertext size(MB)	38.51	33.68	31.95	385.10
Encryption time(s)	12.86	15.70	24.81	126.04

Table 1: Encryption time and ciphertext size of Superfast C-FE scheme: As, encryption time and ciphertext size in our scheme depend only on the plaintext size, we present it separately from Table 2. Experiments were performed on a **laptop** with Intel Core i7 3615QM processor, 8GB memory and Mac OS X 10.9. Each measurement is an average of 10 runs. We evaluated 4,000,000 and 40,000,000

[7]The number is based on the fact that each human has 4,000,000 variants.

[8]4 million variants in each individual can appear at 40 million different locations.

We present the performance of Superfast scheme in Table 1 and Table 2. Function vector is chosen randomly to account for worst case scenarios. As, we pack several secret shares in each public key encryption by concatenating them in a single block, sequential function vectors would result in a very small number of public key decryptions at the authority for key generation. Our plaintext has millions of elements, so, very small random function vectors (e.g., with 1000 elements) would result in number of public key decryptions same as the size of the function vector. But, as the function vector size increases the number of public key decryptions decreases. The maximum possible size of the function vector is when it is equal to the size of plaintext. As, shown in Table 2, this case is much efficient than small function vectors. Note that in practical scenarios function vectors are not random and our scheme will perform much better than this analysis. The key request message size depends upon the function vector size. Function key size is constant in our scheme. Decryption stage is very efficient and constitutes simple additions and multiplications and only depends upon the plaintext size.

6.2 General C-FE scheme

Our general construction is based on Yao's garbled circuits. We use FastGC [35] for Yao's garbled circuits implementation. Our general C-FE construction also requires hybrid encryption, which we implement using AES (with 128-bit key) and RSA-OAEP (with 4096-bit key). To implement hybrid encryption we use Java Cryptography Extension (JCE). For AES, we use AES counter-mode implementation of JCE, and for RSA we use RSA-OAEP implementation of JCE. RSA-OAEP is a non-malleable encryption scheme secure against IND-CCA2 attacks in the random oracle model [16]. Moreover, our prototype is single-threaded. We implement the alternate construction (**with function hiding**) described in Appendix A, which is slightly more efficient. This construction requires minor modifications to the garbled circuit implementation, so in principle it is very easy to reproduce our results. We require extra code for *(i)* hybrid encryption, and *(ii)* partial reuse of the wire labels.

Experiments. We tested our general C-FE construction on a Dell PowerEdge R720 computer with dual Intel Xeon E5-2670 (2.60GHz) processors (computer has 16 cores in total, but our implementation is sequential) and 128GB of memory. Scientific Linux 6.3 (kernel version: 2.6.32) was installed on the computer. We allocated 30GB heap memory for the garbled circuit generator and another 30GB heap memory for the garbled circuit evaluator. We note that FastGC Java implementation is memory intensive and even with 30GB heap memory (maximum allowable by JVM is less than 32GB), we ran out of memory. Kreuter et al. introduced PCF (Portable Circuit Format) framework which is much better in terms of memory consumption, partially because PCF is written in C++ [45]. Both generator and evaluator programs were executed simultaneously on the same machine. Amount of data transfered between generator and evaluator was recorded to report network bandwidth usage.

We evaluated performance of our system on very large problem instances. As shown in Table 3 performance of our scheme is negligible over plain garbled circuits. We evaluated the general scheme with *function hiding* capability. It is clear from Table 3 that our scheme has negligible overhead on top of plain garbled circuits. Note that **Offline** computation time is a one-time cost that's incurred first time the computation is done and there is no computation cost when the same computation is repeated again. The offline computation includes generating plaintext digital circuit from the code.

Key size (bits)	Key generation				Decryption
	Function vector size (4 byte alphabet)	Time (s)	Data sent (KBytes)	Data received (Bytes)	Time (ms)
1024	1,000	1.21	132.46	8	0.07
2048	1,000	6.55	256.21	8	0.07
4096	1,000	42.52	501.26	8	0.07
1024	10,000	11.66	1295.96	8	0.04
2048	10,000	61.95	2422.85	8	0.04
4096	10,000	377.23	4414.68	8	0.04
1024	20,000	22.37	2529.26	8	0.07
2048	20,000	116.83	4551.33	8	0.08
4096	20,000	658.67	7732.90	8	0.17
1024	4,000,000	226.19	55059.63(=53.77MB)	8	20.58
2048	4,000,000	495.31	50118.00(=48.94MB)	8	22.81
4096	4,000,000	1512.44	483444.50(=47.21MB)	8	23.03
1024	40,000,000	1947.32	240209.11(=234.60MB)	8	19.90

Table 2: Performance evaluation of our Superfast inner-product construction: Experiments were performed on a **laptop** with Intel Core i7 3615QM processor, 8GB memory and Mac OS X 10.9. Each measurement is average of 10 runs.

Problem	Encryption		Keygen				Decryption
	Time(ms)	Time(ms)	Decryption(ms)		GC Gen(s)	Comm	GC Eval(s)
	1024	4096	1024	4096	Offline[a], **Online**		Offline[a],**Online**
Hamming 10,000bits	2.87	3.74	5.56	59.30	3.81, **0.30**	919.09KB	2.19, **0.31**
Hamming 16,000bits	4.15	5.15	7.27	61.09	3.95, **0.47**	1470.61KB	2.70, **0.45**
Hamming 20,000bits	5.16	6.09	8.10	61.88	6.19, **0.55**	1838.46KB	4.70, **0.52**
Hamming 60,000bits	14.88	15.66	18.12	72.92	11.15, **1.52**	5515.99KB	88.34, **0.44**
Hamming 1,500,000bits	364.33	379.27	408.94	453.59	469.78, **45.81**	135MB	426.50, **44.26**
Levenshtein 100x100 (2-bit alphabet)	0.35	1.52	3.75	57.22	1.47, **3.47**	10725.26KB	0.59, **3.50**
Levenshtein 1000x1000 (2-bit alphabet)	0.90	1.75	3.96	55.87	1.57, **498.38**	1534MB	0.64, **498.40**
Levenshtein 20000x50 (2-bit alphabet)	0.36	2.46	1.00	55.80	1.64, **650.73**	2098MB	0.70, **650.45**
Levenshtein 20000x200 (2-bit alphabet)	0.35	1.10	2.17	55.31	1.24, **2714.76**	8402MB	0.80, **2714.75**
SmithWaterman 50x50 (32-bit alphabet)	0.35	1.05	2.55	56.03	2.47, **197.51**	580MB	1.58, **197.54**
AES (16 Bytes)	0.50	1.05	3.73	56.91	1.50, **0.17**	306.33KB	0.64, **193.10**
Dot Product 100x100 (8- & 2-bit alphabets)	0.34	1.04	2.62	55.55	2.62, **51.80**	61.78KB	1304, **77.40**
Dot Product 1000x1000 (8- & 2-bit alphabets)	0.79	1.47	2.94	56.09	60.59, **0.29**	617.76KB	592.51, **0.33**
Dot Product 2500x2500 (8- & 2-bit alphabets)	1.70	2.24	4.12	57.15	531.46, **1.08**	1700.62KB	686.71, **1.11**

Table 3: Performance evaluation of our general Controlled Functional Encryption (C-FE) with **function-hiding** construction: Experiments were performed on a Dell R720 computer with dual Intel Xeon E5-2670 (2.60GHz) processors, 128GB of memory and Scientific Linux 6.3. Each measurement is average of 10 runs. Note the negligible performance overhead due to hybrid encryption on top of plain garbled circuits. Evaluator's computation is only $\frac{1}{4}$th of the generator's computation. Both evaluator and generator are taking approx. the same time, because due to pipelining, evaluator is waiting for the partial circuit to be generated and delivered before it can evaluate it. 1024 and 4096 in the third row of the table shows the public key sizes (in bits) used for hybrid encryption.

[a]The offline cost shown in the table is a **one-time cost** and is only incurred for the first time and does *not* need to be repeated for every computation of the same function. It can be done well before the online phase.

Hamming distance. We evaluated our schemes with Hamming distance problems of size upto 1.5 million bits. Most of the commercial services such as 23andme provides human SNP profile with 0.5 million SNPs (recently they started to provide 1 million SNPs). To compare two SNP profiles, direct Hamming distance doesn't work as each SNP is of two bits. We designed the following simple encoding after which we can use Hamming distance to compute the similarity: SNP can have value of either 0, 1 or 2, we represent 0 as 001, 1 as 010 and 2 as 100. After this encoding, computing Hamming distance will give us number of common SNPs between two SNP profiles. As, we represent each SNP with 3 bits, for a 0.5 million SNP profile, we require 1.5 million bits. Therefore, we conducted experiments with 1.5 million bits and our results shows that it requires less than 8 minutes and 135MB network communication to find similarity between two SNP profiles of 0.5 million SNPs each (using our encoding scheme). Note that our Superfast scheme can also be used for computing Hamming distance, but it doesn't provide function hiding, the general scheme provides function hiding but takes more time.

Levenshtein Distance. Levenshtein distance is also a commonly used similarity measure and is computationally much more involved than Hamming distance. Levenshtein distance is computed using dynamic programming algorithm. We ran relatively large problem instance; finding Levenshtein distance between 20,000 and 200 letters strings (from a 2-bit alphabet). Levenshtein distance requires a lot of time and bandwidth but this is due the inefficiency of underlying garbled circuits.

Inner-product. We implemented inner-product functionality into FastGC family to compare the performance of our Superfast C-FE scheme and general C-FE scheme. We used simple modular multiplier circuit [2] to realize multipliers for our inner-product implementation. Inner-product computation in our general model incurs large one-time cost, but is efficient in the online stage. As, can be seen in the last row of Table 3, inner-product have reasonable computation and communication cost.

Other problems: We also tested our general function C-FE scheme on other problems such as AES and SmithWaterman score. Results are shown in Table 3.

7. APPLICATIONS

Personalized Medicine. Personalized medicine is a revolutionary concept in healthcare. Different from a "one-size-fits-all" approach, it enables physicians to prescribe medicine based on the

patients' genomic build-up. Several cryptographic protocols have been proposed for personalized medicine [13, 25]. These protocols are inefficient and incur very high computation and communication costs. Recently, additive homomorphic encryption based protocols have been proposed [10, 11, 12]. These schemes are relatively efficient, but they are very interactive. They require the patient to be online and possess a computer that will be used during a disease susceptibility test. This makes them more difficult to deploy in practice. More seriously, patient computers could get compromised, resulting in leaking their genomic data. We show that our Superfast C-FE schemes can be used to achieve a much more practical scheme: it doesn't require any direct interaction with the patient[9], is much more computationally and storage efficient, and securer as we are using non-malleable public-key encryption as opposed to homomorphic encryption which allows the ciphertext to be arbitrarily modified.

Using our technique, DNA is first digitized through sequencing or genotyping by an external agency. This sequencing agency can encrypt the patient's genome under our Superfast C-FE scheme with a public key issued by the authority and then publish the ciphertext. Later, when a medical unit wants to do some disease susceptibility test, it obtains the encrypted genome and asks the authority for a one-time function key corresponding to the required disease-susceptibility test. The maximum number of disease markers for a disease susceptibility test are no more than 50 Single Nucleotide Polymorphisms (single nucleotide variation between two species) (SNPs) [6]. We conducted our experiments with 1000 SNPs disease test to show the efficiency of our scheme. As shown in Table 2, our scheme can compute disease susceptibility tests very efficiently.

Patient Similarity. Suppose Alice is suffering from a cancer and her physician wants to search (on a nation-wide scale) for another patient with similar symptoms and genetic build-up treated for the same cancer, in a hope that if some therapy and treatment worked or didn't work, it would help to treat Alice's cancer. Hospitals are typically reluctant to share data with each other without proper security protection, due to concerns about privacy and liability. Putting effective protection in place is highly nontrivial, given the scale of the problem: there are 5723 registered hospitals in United States [3] and more than 15 million patients suffering from cancer in US alone [1]. It can be difficult for hospitals to even share the data such as the total number of diabetic patients to form cohorts for medical studies. In this situation, sharing genomic data is extremely far-fetched.

Many schemes [40, 20, 26, 58, 18, 9, 59] have been proposed for measuring similarity between genomes but they are designed for comparing two genomes and none of them can actually gracefully scale to handle the complete human SNP profile similarity comparison (i.e. 4 million letters of 2-bit alphabet). Comparing one SNP profile to potentially thousands or hundred of thousands is out of question for current schemes. We show that our Superfast C-FE scheme can efficiently support complete human SNP profile comparison and is highly scalable to be used for comparison with a very large population.

Each human has 4 million SNPs. Comparing similarity of the complete 4 million SNP profile requires 226 seconds in our scheme and is highly parallelizable. Assuming that the pricing model of the authority is similar to Amazon EC2, similarity comparison of the complete genome would cost only $0.014 per single 4 million SNPs profile comparison. Most of the time complete SNP profile comparison is not required and that's why we also conducted experiments with small function vector size (10,000 and 20,000). Comparison with function vector of size 20,000 can be done in 22.37 seconds and it would cost $0.0014 per comparison. Finding a similar genome in a collection of 100,000 genomes would cost only $1414 when comparing all 4 million SNPs, while it would cost only $140 when comparing any random 20,000 SNPs.

Paternity and Kinship. For privacy-preserving paternity and kinship tests, Baldi et al. make use of both cryptographic tools (i.e., private set intersection) and biological tools (e.g., emulating the Restriction Fragment Length Polymorphism chemical test in software) [13]. Furthermore, their subsequent work demonstrates a framework for conducing such tests on a Android smartphone [25]. Baldi et al. have a very elegant idea of exploiting domain knowledge to bring privacy-preserving genomic computation to the world of plausibility. Their scheme is based on private-set intersection protocol and is not general enough to be used for other types of genomic computation. Moreover, they assume that user is storing her own genome which is not a very practical assumption. Also, their scheme requires access to the complete genome (i.e. 3 billion letters), which is very expensive. Moreover, they only show how they can find paternity test, we can also support other relations such as sibling, uncle, cousin, etc. Our approach can support paternity and kinship inference using human SNP profile that can be obtained for less than $100 (e.g., from 23andme). Our constructions can be used to implement kinship inference algorithms described in [49].

8. RELATED WORK

In this section, we talk about the work most closely related to ours, namely functional encryption schemes that support any polynomial-time computation.

Sahai et al. proposed single-query functional encryption scheme that supports any polynomial-time computation on the ciphertext *only once* [56]. Their construction is based on Yao's garbled circuits [60]. In the encryption phase, a garbled circuit is generated with input of the encryption party embedded in the circuit. Each input wire label of this garbled circuit is encrypted using a different public key. The garbled circuit and encryption of the wire labels are sent to the decryption party. Decryption party asks the authority for the decryption keys of the wire labels corresponding to the function to be computed and it decrypts the wire labels using these keys After obtaining the wire labels the decryption party can evaluate the garbled circuit to compute the function. As they need to encrypt each wire label with a different public key, packing is not possible and ciphertext size blows up significantly. Typically, 80 bit wire labels are used in Yao's garbled circuit implementation and encrypting each wire label with public key will blow up the ciphertext size by $\frac{keysize}{80}\times$ the plaintext size. Moreover, the limitation of computing only single function makes the scheme not useful for many practical applications. In fact, the scheme was presented as a public key encryption scheme, where encryptor can encrypt ciphertext without worrying about the credentials of the receiver. The receiver can only decrypt if it has appropriate credentials. The limit of computing single function comes from the reusability issue of Yao's garbled circuit. Feasibility of reusable garbled circuit has been shown by it very inefficient to be of any practical use [30].

Gorbunov et al. proposed a scheme based on Sahai et al. scheme that supports computation of q-functions over the ciphertext, where q depends on different parameters and increasing q affects the overall efficiency of the scheme [31]. They addresses the reusability is-

[9]If patient wishes, she can opt to be asked by the authority for permission to conduct test using email, SMS, phone call or some other method; alternatively patient can decide beforehand which parties are allowed to learn which functions.

sue of garbled circuit in Sahai et al. construction but this makes the scheme very inefficient. Even with overwhelming overhead the scheme supports limited number of functions to be computed. While theoretically, interesting the scheme is very far from being practical and would take hundreds of years even for very small values of q (e.g., 100).

Chung et al. proposed a functional encryption scheme based on stateless hardware tokens that are identical for all users [24]. However, they rely on computationally intensive operations (fully homomorphic encryption schemes) as well as a powerful tamper proof token carrying out significant computation (signature, non-interactive zero knowledge proofs (NIZK) and succinct non-interactive arguments (SNARGs) verification). Moreover, it has been shown that secrets can be easily stolen from tamper-proof hardware [7], so that a single token can be attacked to compromise the entire system. Overall, it is not clear if token based approach would be practical in various applications.

9. CONCLUSION

Functional encryption has emerged as a fascinating primitive in cryptography recently. We have proposed a complementary variant — called *controlled functional encryption* (C-FE) — which allows a more fine-grained access control than FE allows, but relies on a semi-trusted authority. Our proposal is motivated by several applications where C-FE provides a natural solution. Further, we give efficient constructions for C-FE, relying only on well-known cryptographic assumptions. We evaluate the performance of our C-FE schemes on large scale data and demonstrate its efficiency.

10. ACKNOWLEDGMENTS

This work was partially supported by CNS-1228856, CNS-1017782, CNS-1117106, CNS-1223477, CNS-1223495, CNS-1330491 (THaW), and HHS 90TR0003-01 (SHARPS). The views expressed are those of the authors only.

11. REFERENCES

[1] Cancer facts and statistics. *http://www.cancer.org/research/cancerfactsstatistics/*.

[2] A combinational multiplier using the xilinx spartan ii fpga. *http://ecen3233.okstate.edu/PDF/Labs/Combinational%20Multiplier.pdf*.

[3] Fast facts on US hospitals. *http://www.aha.org/research/rc/stat-studies/fast-facts.shtml*.

[4] Havasupai tribe and the lawsuit settlement aftermath. *http://genetics.ncai.org/case-study/havasupai-Tribe.cfm*.

[5] Indian tribe wins fight to limit research of its dna. *http://www.nytimes.com/2010/04/22/us/22dna.html?pagewanted=all&_r=1&*.

[6] List of genetic diseases with associated genes and snp's. *http://www.eupedia.com/genetics/genetic_diseases.shtml*.

[7] Tpm reset attack. *http://www.cs.dartmouth.edu/~pkilab/sparks/*.

[8] S. Agrawal, S. Gurbanov, V. Vaikuntanathan, and H. Wee. Functional encryption: New perspectives and lower bounds. In *Crypto*, 2013.

[9] M. J. Atallah and J. Li. Secure outsourcing of sequence comparisons. *International Journal of Information Security*, 4(4):277–287, 2005.

[10] E. Ayday, J. L. Raisaro, and J.-P. Hubaux. Privacy-enhancing technologies for medical tests using genomic data. In *NDSS*, 2013.

[11] E. Ayday, J. L. Raisaro, P. J. McLaren, J. Fellay, and J.-P. Hubaux. Privacy-preserving computation of disease risk by using genomic, clinical, and environmental data. In *HealthTech*, 2013.

[12] E. Ayday, J. L. Raisaro, J. Rougemont, and J.-P. Hubaux. Protecting and evaluating genomic privacy in medical tests and personalized medicine. In *WPES*, 2013.

[13] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *CCS*, pages 691–702, 2011.

[14] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO '98*, number 1462, pages 26–45.

[15] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *CCS*, pages 784–796, 2012.

[16] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *EUROCRYPT*, pages 92–111, 1995.

[17] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE S&P*, pages 321–334, 2007.

[18] M. Blanton, M. J. Atallah, K. B. Frikken, and Q. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *ESORICS*, pages 505–522. 2012.

[19] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, pages 253–273, 2011.

[20] F. Bruekers, S. Katzenbeisser, K. Kursawe, and P. Tuyls. Privacy-preserving matching of DNA profiles. *IACR Cryptology ePrint Archive*, 2008:203, 2008.

[21] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[22] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *SCN*, 2013.

[23] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *USENIX Security*, pages 289–304, 2013.

[24] K.-M. Chung, J. Katz, and H.-S. Zhou. Functional encryption from (small) hardware tokens. In *ASIACRYPT*, pages 120–139. 2013.

[25] E. De Cristofaro, S. Faber, P. Gasti, and G. Tsudik. Genodroid: are privacy-preserving genomic tests ready for prime time? In *WPES*, pages 97–108, 2012.

[26] D. Eppstein, M. T. Goodrich, and P. Baldi. Privacy-enhanced methods for comparing compressed DNA sequences. *arXiv preprint arXiv:1107.3593*, 2011.

[27] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *CRYPTO*, number 2139, pages 260–274. Jan. 2001.

[28] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *STOC*, pages 40–49, 2013.

[29] S. Garg, C. Gentry, S. Halevi, A. Sahai, and B. Waters. Attribute-based encryption for circuits from multilinear maps. In *CRYPTO 2013*, number 8043, pages 479–499. 2013.

[30] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *STOC*, pages 555–564, 2013.

[31] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Functional encryption with bounded collusions from multiparty computation. In *CRYPTO*, 2012.

[32] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Functional encryption with bounded collusions via multi-party computation. In *CRYPTO*, pages 162–179. 2012.

[33] S. Gorbunov, V. Vaikuntanathan, and H. Wee. Attribute-based encryption for circuits. In *STOC*, pages 545–554, 2013.

[34] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, pages 89–98, 2006.

[35] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols. In *NDSS*, 2012.

[36] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, volume 201, 2011.

[37] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE S&P*, pages 272–284, 2012.

[38] Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *CRYPTO*, pages 18–35. 2013.

[39] Y. Ishai. Randomization techniques for secure computation. *Secure Multi-Party Computation*, 10:222–248, 2013.

[40] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *IEEE S&P*, pages 216–230, 2008.

[41] A. Joux. Faster index calculus for the medium prime case application to 1175-bit and 1425-bit finite fields. In *EUROCRYPT*, pages 177–193, 2013.

[42] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.

[43] S. Kamara, P. Mohassel, and B. Riva. Salus: a system for server-aided secure function evaluation. In *CCS*, pages 797–808, 2012.

[44] J. Katz, A. Sahai, and B. Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *EUROCRYPT*, pages 146–162, 2008.

[45] B. Kreuter, B. Mood, A. Shelat, and K. Butler. Pcf: A portable circuit format for scalable two-party secure computation. *USENIX Security*, 2013.

[46] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, pages 14–14, 2012.

[47] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, pages 1–17. 2013.

[48] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay-secure two-party computation system. In *USENIX Security*, pages 287–302, 2004.

[49] A. Manichaikul, J. C. Mychaleckyj, S. S. Rich, K. Daly, M. Sale, and W.-M. Chen. Robust relationship inference in genome-wide association studies. *Bioinformatics*, 26(22):2867–2873, 2010.

[50] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. Privacy-preserving matrix factorization. In *CCS*, pages 801–812, 2013.

[51] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*, pages 334–348, 2013.

[52] T. Okamoto and K. Takashima. Fully secure functional encryption with general relations from the decisional linear assumption. In *CRYPTO 2010*, number 6223, pages 191–208. 2010.

[53] A. O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010.

[54] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008*, number 5157, pages 554–571. 2008.

[55] M. Prabhakaran and M. Rosulek. Rerandomizable rcca encryption. In *CRYPTO*, pages 517–534, 2007.

[56] A. Sahai and H. Seyalioglu. Worry-free encryption: functional encryption with public keys. In *CCS*, pages 463–472, 2010.

[57] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.

[58] D. Szajda, M. Pohl, J. Owen, B. G. Lawson, and V. Richmond. Toward a practical data privacy scheme for a distributed implementation of the smith-waterman genome sequence comparison algorithm. In *NDSS*, 2006.

[59] R. Wang, X. Wang, Z. Li, H. Tang, M. K. Reiter, and Z. Dong. Privacy-preserving genomic computation through program specialization. In *CCS*, pages 338–347, 2009.

[60] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

[61] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *IEEE S&P*, pages 493–507, 2013.

APPENDIX

A. ALTERNATE GENERAL CONSTRUCTION

Before a formal description, we give some intuition about the construction. At a high-level, our plan is to let the client use a garbled circuit generated by the authority to evaluate the function on the input x. The circuit in question evaluates the function F, which takes x and f as inputs (and optionally, a tweak w, which we ignore for simplicity). f is known to the client, and in the function-revealing case, to the authority as well. But note that neither the client nor the authority knows the input x (which was generated by a data owner).[10] Thus, it is not clear how a garbled circuit for F can be used in our setting.

In our solution, the data owner will arrange for the client to have the labels corresponding to the input x, without either the client or the authority knowing x. The key idea is that the authority does not pick all the labels for all the wires in the garbled circuit itself. Instead, the data owner would specify the labels used for the input wires corresponding to input x. More precisely, suppose x is k bits long. Then the data owner picks $2k$ random labels $\left\{r_0^i, r_1^i\right\}_{i=1}^k$ and encrypts them (using a CCA2 secure public-key encryption scheme) for the authority. The encryption is required because it is important that these labels are not all known to the client. During the key-request, the client is expected to send this part of the ciphertext to the authority. While a garbled circuit is generated, for each wire u in the circuit, two freshly chosen random labels $(R_0^{(u)}, R_1^{(u)})$ are required; but for the wires corresponding to the input x_i (i.e., the i^{th} bit of x), the authority uses the pair (r_0^i, r_1^i) instead. (Labels for all the other wires are picked freshly.) Now, the data owner knows both x and the labels for the input wires used in the garbled circuit. So it can simply provide the correct labels to the client as part of the encryption of x. More precisely, the client will be given the k labels $r_1^{x_1}, \ldots, r_k^{x_k}$. On obtaining the garbled circuit, it simply evaluates the garbled circuit using these labels for the wires corresponding to x.

We remark that typically, it is important that a garbled circuit is not reused – i.e., evaluated on different inputs. Our solution could be viewed as a safe way of reusing *parts* of a garbled circuit. In particular, if different functions are evaluated on the same input x, the same labels can be used for x. (In a standard 2-party computation, this observation could be used to replace multiple OT protocol invocations, with a single OT protocol; in our case, since the data owner is offline, this property is crucial.)

Protocol Π'

Since Π' is a slight modification of Π, we only describe what changes need to be made to the algorithms of Π in order to obtain the ones for Π'.

- $\mathsf{Setup}'(1^\kappa)$: Stays the same as Setup.
- $\mathsf{Enc}'(x, \mathsf{MPK})$: Randomly pick $2k$ bit strings of length κ each. Let (r_i^0, r_i^1) denote the ith pair among them, where $1 \leq i \leq k$. Let $r_x = r_1^{x_1} \circ r_2^{x_2}, \ldots \circ r_k^{x_k}$, where x_i denotes the ith bit of x. Also, let $\hat{r} = r_1^0 \circ r_1^1 \circ r_2^0 \circ r_2^1 \ldots \circ r_k^0 \circ r_k^1$ be the concatenation of all the randomly chosen strings (in the specified order). Output the ciphertext $\mathsf{CT}_x = (r_x, \mathsf{Enc}_{\mathsf{CCA2}}(\hat{r}, \mathsf{MPK}))$.
- $\mathsf{KeyReq}'(\mathsf{CT}, f)$: Stays the same as KeyReq.
- $\mathsf{KeyGen}'(\rho = (\sigma, M), \mathsf{MSK})$: Run $\mathsf{Dec}_{\mathsf{CCA2}}(\sigma, \mathsf{MSK})$ to obtain $(r_1^0, r_1^1), \ldots, (r_k^0, r_k^1)$. Consider a circuit C which takes as input a k-bit string z and an ℓ-bit function g, and computes $F(g, z)$. Construct a garbled circuit $\hat{C}$ for C, but with (r_i^0, r_i^1) as keys for the input bit z_i ($i \in [1, k]$). All the other keys required for creating the garbled circuit are chosen at random. M' is computed in the same way as KeyGen. Output $\tau = (\hat{C}, M')$.
- $\mathsf{Dec}'(\zeta, \tau)$: Key for the jth bit of g ($j \in [1, \ell]$) is obtained in the same way as described in Dec. On the other hand, key for the ith bit of z is part of ζ. Now, evaluate $\hat{C}$ to obtain the value of $f(x)$.

Figure 4: Alternate C-FE Construction

[10] Further, as we consider malicious client which may attempt to alter the input on which the garbled circuit is evaluated, which rules out a simple solution in which x is kept secret-shared between the client and the authority (unless, cryptographic operations are incorporated into the garbled circuit).