# Fully Homomorphic Message Authenticators

Rosario Gennaro[1] Daniel Wichs[2]

[1]The City College of CUNY

[2]Northeastern University

Asiacrypt 2013

# Our Results

## Fully Homomorphic MACs

A way to authenticate data so that the result of *any* function $F$ can also be authenticated as the correct evaluation of $F$ over the authenticated data

## What does it mean

A *client* stores some data with a *server* $D$ and a MAC $T_D$ on $D$ computed using a *short* secret key $sk$. When queried on a function $F$, the server returns $y = F(D)$ and a *short* tag $T_y$ which can be verified as correct using $sk$.

# Our Results

## Fully Homomorphic MACs

A way to authenticate data so that the result of *any* function $F$ can also be authenticated as the correct evaluation of $F$ over the authenticated data

## What does it mean

A *client* stores some data with a *server* $D$ and a MAC $T_D$ on $D$ computed using a *short* secret key $sk$. When queried on a function $F$, the server returns $y = F(D)$ and a *short* tag $T_y$ which can be verified as correct using $sk$.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades". Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$ and Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$. Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$, and
  - Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$
  - Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$, and
  - Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$
  - Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$, and
  - Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$
  - Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$, and
  - Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$
  - Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$.

# Labeled Data and Programs

- To differentiate data (and the computations performed on it) we rely on *labels*;
- When the client authenticates $D$, she chooses a label $\tau$ for it, which is given as input to the authentication algorithm;
- Correspondingly we consider *labeled programs* $P$, where each input of the program has an associated label $\tau$ indicating which data it should be evaluated on.
  - Think of labels as a filenames, e.g. "Class Grades"
  - Later the query is "Please return the average of Class Grades"
- Homomorphic authenticators allow to authenticate the output of a labeled program, given authentication tags for correspondingly labeled input data.
  - Given a labeled program $P$ on $n$ inputs whose labels are $\tau_1, \ldots, \tau_n$, and
  - Data $D_1, \ldots, D_n$ with authentication tags $T_1, \ldots, T_n$
  - Anybody can homomorphically compute a short authentication tag $T_y$ that authenticates the output $y = P(D_1, \ldots, D_n)$.

# Composition

- We construct homomorphic authenticators that are *composable* i.e. the result $y$ and its computed tag $T_y$ can be used as input to another homomorphic evaluation.
  - Assume that the tags $T_1, \ldots, T_n$ authenticate some data $y_1, \ldots, y_n$ as the outputs of some labeled programs $P_1, \ldots, P_n$
  - Let $P^*$ be an $n$-input labeled program then anybody can compute $T^*$ that authenticates $y^* = P^*(y_1, \ldots, y_n)$ using only the pairs $(y_i, T_i)$.

# Composition

- We construct homomorphic authenticators that are *composable* i.e. the result $y$ and its computed tag $T_y$ can be used as input to another homomorphic evaluation.
  - Assume that the tags $T_1, \ldots, T_n$ authenticate some data $y_1, \ldots, y_n$ as the outputs of some labeled programs $P_1, \ldots, P_n$
  - Let $P^*$ be an $n$-input labeled program then anybody can compute $T^*$ that authenticates $y^* = P^*(y_1, \ldots, y_n)$ using only the pairs $(y_i, T_i)$.

# Composition

- We construct homomorphic authenticators that are *composable* i.e. the result $y$ and its computed tag $T_y$ can be used as input to another homomorphic evaluation.
  - Assume that the tags $T_1, \ldots, T_n$ authenticate some data $y_1, \ldots, y_n$ as the outputs of some labeled programs $P_1, \ldots, P_n$
  - Let $P^*$ be an $n$-input labeled program then anybody can compute $T^*$ that authenticates $y^* = P^*(y_1, \ldots, y_n)$ using only the pairs $(y_i, T_i)$.

# Related Work

- Homomorphic authenticators (and signatures) for *linear* functions were constructed for the application of *network coding* (starting from [JMSW02])
- Homomorphic signatures for *polynomials* were presented in [BF11]
- We were the first to consider *arbitrary* functions
- Following our work, [CF13] shows a very nice and simple approach for functions $F$ described by *shallow* arithmetic circuits.

# Related Work

- Homomorphic authenticators (and signatures) for *linear* functions were constructed for the application of *network coding* (starting from [JMSW02])
- Homomorphic signatures for *polynomials* were presented in [BF11]
- We were the first to consider *arbitrary* functions
- Following our work, [CF13] shows a very nice and simple approach for functions $F$ described by *shallow* arithmetic circuits.

# Related Work

- Homomorphic authenticators (and signatures) for *linear* functions were constructed for the application of *network coding* (starting from [JMSW02])
- Homomorphic signatures for *polynomials* were presented in [BF11]
- We were the first to consider *arbitrary* functions
- Following our work, [CF13] shows a very nice and simple approach for functions $F$ described by *shallow* arithmetic circuits.

# Related Work

- Homomorphic authenticators (and signatures) for *linear* functions were constructed for the application of *network coding* (starting from [JMSW02])
- Homomorphic signatures for *polynomials* were presented in [BF11]
- We were the first to consider *arbitrary* functions
- Following our work, [CF13] shows a very nice and simple approach for functions $F$ described by *shallow* arithmetic circuits.

# Related Work: SNARGs

**Succint Non-interactive Arguments** [M94] can be used to produce *short* proofs that a certain computation is correct.

Unfortunately SNARGs must rely on non-standard assumption (e.g. random oracle, or knowledge assumptions) [GW11], while our construction relies only on FHE and PRF security.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

**Secure Memory Delegation** Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

Secure Memory Delegation Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

**Secure Memory Delegation** Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

**Secure Memory Delegation** Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

**Secure Memory Delegation** Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

# Related Work: Secure Delegation

**Delegation of Computation:** The *dual* problem in which the client authenticates the *function* $F$ to the server, and then queries it on the input $D$ to get an authenticated result $y = F(D)$ [GGP10]. Can be used by outsourcing a *universal* circuit $C_D$ which has the data $D$ hard-wired in it and the function $F$ is the input. While this approach yields efficient verification, it also:

- requires interaction: client must create a challenge for $F$;
- bounds the size of the function $F$;
- does not yield a composable scheme;
- requires the data $D$ to be authenticated in one-shot.

**Secure Memory Delegation** Presented first by [CKLR11] is composable and has fast verification. However it is still interactive, and also requires the client to maintain state.

## Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0; PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \dots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \dots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0; PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \dots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \dots, c_n$;
- To verify $y, \gamma_1, \dots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \dots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \dots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \dots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \dots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \dots, \gamma_k$ the client checks:

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \dots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \dots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \dots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \dots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \dots, \gamma_k$ the client checks:
  - If $i \in S$ that $c_i$ decrypts to $y$;
    If $i \notin S$, the client can recompute $c_i$ (does not depend on data, only on short PRF key $K$) and apply the FHE evaluation procedure and check if it outputs $\gamma_i$.

## Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \dots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \dots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \dots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \dots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \dots, \gamma_k$ the client checks:
  - If $i \in S$ that $c_i$ decrypts to $y$;
  - If $i \notin S$, the client can recompute $c_i$ (does not depend on data, only on short PRF key $K$) and apply the FHE evaluation procedure and check if it outputs $\gamma_i$.

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:
  - If $i \in S$ that $c_i$ decrypts to $y$;
  - If $i \notin S$, the client can recompute $c_i$ (does not depend on data, only on short PRF key $K$) and apply the FHE evaluation procedure and check if it outputs $\gamma_i$.

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:
  - If $i \in S$ that $c_i$ decrypts to $y$;
  - If $i \notin S$, the client can recompute $c_i$ (does not depend on data, only on short PRF key $K$) and apply the FHE evaluation procedure and check if it outputs $\gamma_i$.

# Birds-eye View Of Our Construction

- Our construction uses *Fully Homomorphic Encryption* (FHE) and Pseudo-Random Functions (PRF). For security parameter $k$, to generate a key the client will choose
  - $pk$ an FHE public key;
  - $K$ a PRF secret key;
  - and a subset $S$ of $[1 \ldots k]$ of size $k/2$;
- To authenticate data $D$, the client will produce as a tag $k$ ciphertexts $c_1, \ldots, c_n$ as follows:
  - If $i \in S$ then $c_i = FHE_{pk}(D; r_i)$ otherwise $c_i = FHE(0, PRF_K(i))$;
- When queried on a function $F$ the server returns $y = F(D)$ and $k$ ciphertexts $\gamma_1, \ldots, \gamma_k$ computed by evaluating $F$ over the ciphertexts $c_1, \ldots, c_n$:
  - i.e. compute $\gamma_i$ by applying the FHE evaluation procedure for $F$ to $c_i$;
- To verify $y, \gamma_1, \ldots, \gamma_k$ the client checks:
  - If $i \in S$ that $c_i$ decrypts to $y$;
  - If $i \notin S$, the client can recompute $c_i$ (does not depend on data, only on short PRF key $K$) and apply the FHE evaluation procedure and check if it outputs $\gamma_i$.

# Intuition of Security Proof

- Intuitively, the only way that an attacker can lie about the output $y = F(D)$ is by producing a tag $(\hat{\gamma}_1, \ldots, \hat{\gamma}_k)$ where the ciphertexts $\hat{\gamma}_i$ for $i \notin S$ are computed correctly but for $i \in S$ they are all modified so as to encrypt the wrong output.
- This should be hard due to
  - the semantic security of the FHE (hard to tell which ciphertexts encrypt the data and which ones encrypt 0)
  - the security of the PRFs (the ciphertexts creating using pseudo-random coins should be indistinguishable from regular ciphertexts).

# Intuition of Security Proof

- Intuitively, the only way that an attacker can lie about the output $y = F(D)$ is by producing a tag $(\hat{\gamma}_1, \ldots, \hat{\gamma}_k)$ where the ciphertexts $\hat{\gamma}_i$ for $i \notin S$ are computed correctly but for $i \in S$ they are all modified so as to encrypt the wrong output.
- This should be hard due to
  - the semantic security of the FHE (hard to tell which ciphertexts encrypt the data and which ones encrypt $0$)
  - the security of the PRFs (the ciphertexts creating using pseudo-random coins should be indistinguishable from regular ciphertexts).

# Intuition of Security Proof

- Intuitively, the only way that an attacker can lie about the output $y = F(D)$ is by producing a tag $(\hat{\gamma}_1, \ldots, \hat{\gamma}_k)$ where the ciphertexts $\hat{\gamma}_i$ for $i \notin S$ are computed correctly but for $i \in S$ they are all modified so as to encrypt the wrong output.
- This should be hard due to
  - the semantic security of the FHE (hard to tell which ciphertexts encrypt the data and which ones encrypt $0$)
  - the security of the PRFs (the ciphertexts creating using pseudo-random coins should be indistinguishable from regular ciphertexts).

# Intuition of Security Proof

- Intuitively, the only way that an attacker can lie about the output $y = F(D)$ is by producing a tag $(\hat{\gamma}_1, \ldots, \hat{\gamma}_k)$ where the ciphertexts $\hat{\gamma}_i$ for $i \notin S$ are computed correctly but for $i \in S$ they are all modified so as to encrypt the wrong output.
- This should be hard due to
  - the semantic security of the FHE (hard to tell which ciphertexts encrypt the data and which ones encrypt $0$)
  - the security of the PRFs (the ciphertexts creating using pseudo-random coins should be indistinguishable from regular ciphertexts).

# Drawback 1: Verification Time

- The size of the key and the tags is *independent* of the size of the data;
- *However* the client verification time requires time proportional to the computation of $F$ (note that the client has to recompute $\gamma_i$ from $c_i$ using the FHE evaluation procedure for $F$).

# Drawback 1: Verification Time

- The size of the key and the tags is *independent* of the size of the data;
- *However* the client verification time requires time proportional to the computation of $F$ (note that the client has to recompute $\gamma_i$ from $c_i$ using the FHE evaluation procedure for $F$).

# Fast Verification Time

If we are willing to add interaction to our scheme we can obtain fast verification (independent of the size of the program) by *outsourcing* the verification task to the server!

# Drawback 2: Verification Queries

- Our construction is secure in the setting where the attacker cannot make *verification queries* to test whether a maliciously produced tag verifies correctly.
  - Similar to *rejection problem* in FHE-based secure delegation: verification queries provide a decryption oracle for the FHE, which kills semantic security
  - In practice, this means the client must stop using the scheme whenever she gets the first tag that doesn't verify correctly.

# Drawback 2: Verification Queries

- Our construction is secure in the setting where the attacker cannot make *verification queries* to test whether a maliciously produced tag verifies correctly.
  - Similar to *rejection problem* in FHE-based secure delegation: verification queries provide a decryption oracle for the FHE, which kills semantic security
  - In practice, this means the client must stop using the scheme whenever she gets the first tag that doesn't verify correctly.

# Drawback 2: Verification Queries

- Our construction is secure in the setting where the attacker cannot make *verification queries* to test whether a maliciously produced tag verifies correctly.
  - Similar to *rejection problem* in FHE-based secure delegation: verification queries provide a decryption oracle for the FHE, which kills semantic security
  - In practice, this means the client must stop using the scheme whenever she gets the first tag that doesn't verify correctly.

# Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
  - Recall that if $c = FHE_{pk}(D;r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F,c) = FHE(F(D);r^*)$ for *some* randomness $r^*$.
  - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F,r)$
- then our scheme remains secure in the presence of verification queries

# Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
  - Recall that if $c = FHE_{pk}(D; r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F, c) = FHE(F(D); r^*)$ for *some* randomness $r^*$.
  - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F, r)$
- then our scheme remains secure in the presence of verification queries
  - the client will accept only queries where the randomness can be traced to the original randomness
  - rejection of a verification query now does not provide the attacker with a meaningful decryption oracle for the FHE

## Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
  - Recall that if $c = FHE_{pk}(D; r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F, c) = FHE(F(D); r^*)$ for *some* randomness $r^*$.
  - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F, r)$
- then our scheme remains secure in the presence of verification queries
  - the client will accept only queries where the randomness can be traced to the original randomness
  - rejection of a verification query now does not provide the attacker with a meaningful decryption oracle for the FHE

# Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
  - Recall that if $c = FHE_{pk}(D; r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F, c) = FHE(F(D); r^*)$ for *some* randomness $r^*$.
  - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F, r)$
- then our scheme remains secure in the presence of verification queries
  - the client will accept only queries where the randomness can be traced to the original randomness
  - rejection of a verification query now does not provide the attacker with a meaningful decryption oracle for the FHE

# Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
    - Recall that if $c = FHE_{pk}(D; r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F, c) = FHE(F(D); r^*)$ for *some* randomness $r^*$.
    - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F, r)$
- then our scheme remains secure in the presence of verification queries
    - the client will accept only queries where the randomness can be traced to the original randomness
    - rejection of a verification query now does not provide the attacker with a meaningful decryption oracle for the FHE

# Handling Verification Queries

- Assume that the FHE scheme satisfies an additional *randomness homomorphism* property:
  - Recall that if $c = FHE_{pk}(D; r)$ then there is an algorithm $Eval$ such that for any function $F$: $Eval(F, c) = FHE(F(D); r^*)$ for *some* randomness $r^*$.
  - A *randomness homomorphic* FHE also has a second algorithm $REval$ such that $r^* = REval(F, r)$
- then our scheme remains secure in the presence of verification queries
  - the client will accept only queries where the randomness can be traced to the original randomness
  - rejection of a verification query now does not provide the attacker with a meaningful decryption oracle for the FHE

## Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic Signatures: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)

# Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic **Signatures**: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)

# Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic **Signatures**: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)

# Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic **Signatures**: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)

# Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic **Signatures**: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)

# Open Questions

- Do we really need FHE?
- Remove the limitation on Verification Queries
  - Do *randomness homomorphic* FHE schemes exist?
- Fully Homomorphic **Signatures**: add public verification to our scheme;
- Can we obtain verification time independent of $F$? Without interaction and using only standard assumptions?
- Reduce size of tags (right now $O(k)$ where $k$ is the security parameter)