Generalized Non-Interactive Oblivious Transfer Using Count-Limited Objects with Applications to Secure Mobile Agents[*]

Vandana Gunupudi[1] and Stephen R. Tate[2]

[1] Dept. of Computer Science and Engineering, University of North Texas, Denton, TX 76203
vgunupudi@gmail.com
[2] Dept. of Computer Science, University of North Carolina at Greensboro, Greensboro, NC 27402
srtate@uncg.edu

Abstract. Oblivious transfer (OT) is a fundamental primitive used in many cryptographic protocols, including general secure function evaluation (SFE) protocols. However, interaction is a primary feature of any OT protocol. In this paper, we show how to remove the interaction requirement in an OT protocol when parties participating in the protocol have access to slightly modified Trusted Platform Modules, as defined by Sarmenta *et al.* in proposing the notion of count-limited objects (clobs) [8]. Specifically, we construct a new cryptographic primitive called "generalized non-interactive oblivious transfer"(GNIOT). While it is possible to perform GNIOT using clobs in a straightforward manner, with multiple clobs, we show how to perform this *efficiently*, by using a single clob regardless of the number of values that need to be exchanged in an oblivious manner. Additionally, we provide clear definitions and a formal proof of the security of our construction. We apply this primitive to mobile agent applications and outline a new secure agent protocol called the GTX protocol which provides the same security guarantees as existing agent protocols while removing the need for interaction, thus improving efficiency.

1 Introduction

Oblivious Transfer (OT) was introduced by Rabin [7] as a fundamental cryptographic primitive, and subsequently many variants have been studied and used in a variety of cryptographic protocols such as secure multi-party computation. In a 1-out-of-2 OT protocol, Alice (the sender) has 2 values s_0 and s_1, and Bob (the receiver) has a selection bit c. At the end of the protocol, Bob learns the

---

value s_c while obtaining no information about s_{1-c}, and Alice cannot determine which value Bob received. While in some OT variants the selection bit c is random, in this paper we only consider variants in which Bob selects the value c. We call an OT protocol *interactive* if Bob must communicate with Alice or some other party after selecting c, and *non-interactive* otherwise.

In the standard model of computation, non-interactive OT is clearly impossible: Bob can take a "snapshot" of his state immediately before picking a value of c, and then run his computation with $c = 0$ to learn s_0. Since this computation was non-interactive, no state external to Bob is affected, so Bob can roll back his state to the snapshot and re-run his computation with $c = 1$, thus learning s_1 as well.

In this paper we consider a slightly augmented model of computation, reflecting changes happening in real systems with "Trusted Computing" technologies, and show that interactive OT is possible in such a model. We consider how to efficiently accomplish an expanded and generalized form of non-interactive oblivious transfer in such a model, define sensible security properties which we prove hold in our protocols, and explore how this non-interactive oblivious transfer can be used to improve the efficiency of secure function evaluation and secure mobile agent protocols.

Trusted Computing is an initiative of the Trusted Computing Group [12], an industry consortium of over 160 companies, to strengthen security in computing platforms through the use of trusted hardware. Key to Trusted Computing are devices, called Trusted Platform Modules (TPMs) [13], which are already appearing in many desktop PCs and laptops. Various researchers have begun to explore the capabilities of systems that use these hardware modules, utilizing their unique functionality for various real-world applications. Recent work at MIT by Sarmenta *et al.* [8] has introduced the idea of a *virtual monotonic counter* which can be used as a building block for various applications like digital cash, e-wallets, virtual trusted storage and digital rights management (DRM). A virtual monotonic counter is a trusted counter that can be incremented but not reset back to any previous value, thus removing the ability to roll the system back to a previous state as described above. This security property is enforced by the TPM alone and does not require a trusted OS for this purpose — in fact, the required capabilities can be provided by other system augmentations, including smartcards or other crypto processors that control key usage. In addition to having interesting applications, virtual monotonic counters allow us to realize *count-limited objects* or *clobs* which are tied to a particular virtual monotonic counter. Examples of these include *n-time use* decryption or signature keys. The use of each key is tied to a counter which enforces the condition that the key is not used more than n times.

In this paper, we show how to use count-limited objects to implement a useful generalized form of non-interactive oblivious transfer. This new primitive, which we call "Generalized Non-interactive Oblivious Transfer" (GNIOT), is a way of performing a collection of general (k-out-of-n) independent oblivious transfers with a single request. In Section 3 we present a formal definition with the desired

security properties, along with our implementation and a security proof. GNIOT can be accomplished in an obvious and inefficient way by using a distinct clob for each value to be transferred, but this requires a significant number of expensive key generation steps (one RSA key generation per clob). In this paper, we show how to accomplish this in an *efficient* manner — by using a *single* clob, regardless of the number of values to be transmitted. As an example application of GNIOT, we show how this primitive can be directly applied to mobile agent computation, where strong security is often enforced by *interactive* oblivious transfer in various agent protocols. Removing the interaction from these agent protocols removes a significant bottleneck to their efficiency and practicality.

In summary, our contributions include

– Definition of a new primitive called "Generalized Non-interactive Oblivious Transfer", which is impossible to implement in standard computation models, but is possible in a realistically augmented model based on Trusted Computing technologies;
– An implementation of GNIOT which has significantly improved efficiency over the straightforward implementation;
– Careful security analysis and rigorous proofs of our implementation; and
– Use of the GNIOT primitive to create a new *non-interactive*, secure agent protocol called the *GTX* protocol.

2 Definitions and Preliminaries

In this section, we briefly present background information on the building blocks of GNIOT, namely, oblivious transfer and count-limited objects.

2.1 Virtual Monotonic Counters and Count-Limited Objects

Sarmenta *et al.* [8] outline how to create a potentially unlimited number of virtual monotonic counters from a physical monotonic counter or from other potential capabilities of TPMs. While this requires some changes in TPMs, the additional requirements are quite modest, as outlined in this section. They model a virtual monotonic counter as a mechanism that stores a value and provides 2 commands to access this value: a `Read` command that returns the current value of the counter, and an `Increment` command that increments the value of the counter and returns the updated value of the counter. A virtual monotonic counter must be *non-volatile*, i.e., the value of the counter must not change unless incremented in response to a command. It must also be *irreversible*, namely, it must be infeasible for any adversary (including the owner) to reset the counter to any previous value. Finally, the virtual counter must produce verifiable output. This is accomplished by using *unforgeable* execution certificates. First, the counter produces a verifiable output message in response to the Read or Increment commands. This output is then typically signed using an Attestation Identity Key (AIK)[1] and random nonces are used to prevent replay attacks.

---

[1] An AIK is a special type of signature key created on a TPM. The private portion of this key is non-migratable.

Building from these virtual counters, Sarmenta *et al.* have proposed count-limited objects, or clobs, as an interesting and important primitive. These are proposed objects that utilize the ability of a TPM to encrypt data or keys into "blobs" such that they can only be decrypted when the TPM is in a specified state, which in current TPMs is limited to conditions based on the PCRs. In Sarmenta's construction, these encrypted blobs are then linked to a virtual monotonic counter which is used to track/limit the usage of the blob. They also proposed an efficient hash-tree based scheme that allows the TPM to keep track of a large number of virtual monotonic counters, thereby enabling various count-limited objects, each having its own dedicated virtual monotonic counter. While this scheme requires a new command to be added to the TPM, the computations required are relatively simple and could easily be implemented on the microcontrollers that current TPMs are being built from.

2.2   Non-Interactive Oblivious Transfer

In this section we outline new ideas on how count-limited objects can be used to implement a non-interactive version of standard *oblivious transfer*. In an oblivious transfer protocol, two parties can exchange information without learning anything about each other's inputs.

1-out-of-2 Oblivious Transfer (OT): In the standard 1-out-of-2 OT, when Alice transmits one of s_0 or s_1 to Bob in an oblivious manner, interaction between Alice and Bob is typically required. In a common solution, Bob needs to supply Alice with keys to encrypt her strings and this is done only after he decides which value he requires. Therefore, Alice cannot encrypt the strings unless Bob sends her the keys, which he cannot do until he decides which string he wants. Using count-limited objects, Bob can compute keys before making a decision of which s_c he wants, and his later use of that key is restricted by the count-limited property.

We point out that Bellare and Micali [3] have previously introduced a related but different notion of non-interactive oblivious transfer, but in their case Bob receives a randomly selected s_c (he doesn't get to choose which one). This is useful in some applications, but not in the Secure Function Evaluation problems that we are interested in, such as secure mobile agents.

Non-interactive OT using a count-limited decryption key: Alice has 2 values s_0 and s_1. Bob has a TPM and generates a one-time use non-migratable key pair, K_p, K_s and publishes the public key K_p, which is certified using an AIK I_b, which in turn is certified by a Privacy CA. This one-time use key pair is tied to a virtual monotonic counter which limits the private key K_s to being used no more than once. Alice encrypts both values s_0 and s_1 using K_p, having verified that the key is indeed Bob's via the accompanying certificate. At some later time, after receiving the ciphertexts, Bob can decide which value he wants. Then Bob decrypts only that value using K_s, being restricted to do so by the virtual monotonic counter, which is incremented as soon as one of the values is decrypted.

This clearly solves the non-interactive OT problem, but in applications which use multiple oblivious transfers, a separate key must be generated for each OT, which is very inefficient. In the following section, we will show how a *single* clob can control multiple oblivious transfers.

3   Generalized Non-Interactive Oblivious Transfer

We generalize the 1-out-of-2 OT concept to a form where multiple independent oblivious transfers (of the general k-out-of-n type) are defined as part of a single operation. In many applications (such as secure function evaluation) multiple instances of OT must be run, so by defining this as a single operation we have the flexibility of creating solutions which can exploit improvements possible by aggregating multiple requests. We call this combined operation "*generalized non-interactive oblivious transfer* (GNIOT)," which we formally define in the following section.

3.1   Problem Definition

We first define Generalized Oblivious Transfer (GOT), and we will subsequently define phases which will force this to be non-interactive, producing GNIOT.

Definition 1 (GOT). *Define λ as the security parameter and l_d as the length of the data items being sent by Alice to Bob. Assume that Alice has n data sets $S_1, S_2, \cdots, S_n$, with values $x_{i,j} \in \{0,1\}^{l_d}$ for $i \in \{1, 2, \cdots, n\}$ and $j \in \{1, 2, \cdots, m_i\}$, and parameters $k_1, k_2, \ldots, k_n$, where $1 \le k_i \le m_i$. At the end of the GOT execution, Bob will have either no result (represented by $\bot$) or a set of exactly k_i values of his choice from each set S_i, for $i \in \{1, 2, \cdots, n\}$.*

We will need to refer to sets of indices into the data set, so define index set $\mathcal{I}$ to be a set of indices (i, j), and define $\mathcal{I}(i) = \{j \mid (i, j) \in \mathcal{I}\}$. With respect to the parameters provided in an instance of GOT, we say that index set $\mathcal{I}$ is *well-formed* if $|\mathcal{I}(i)| = k_i$ for all $i \in \{1, \ldots, n\}$.

We define GNIOT as a set of operations which perform GOT, but accomplish this task without requiring any interaction between the receiver and another party after the receiver decides which values he wants. For maximum flexibility, allowing either batched or individual decryptions, we define the decryption operation as a stateful process which is called repeatedly — only at the very end are we required to have the actual plaintext values.

Definition 2 (GNIOT). *Generalized Non-Interactive Oblivious Transfer consists of the following phases, which provide a solution to the GOT problem.*

Setup Phase. *This phase involves key generation. Given security parameter λ, the key generation algorithm returns*

$$(\mathcal{K}_p, \mathcal{K}_s) \leftarrow Setup(1^\lambda)$$

where $\mathcal{K}_p$ is the public key information, and $\mathcal{K}_s$ is the secret key information.

Transmit Phase. *This phase transforms the set of values $x_{i,j} \in \{0,1\}^{l_d}$ for $i \in \{1, 2, \cdots, n\}$ and $j \in \{1, 2, \cdots, m_i\}$ into a data blob which can be transmitted to the receiver. Specifically,*

$$C \leftarrow Transmit_{\mathcal{K}_p} \begin{pmatrix} \langle k_1, x_{1,1}, x_{1,2}, \cdots, x_{1,m_1} \rangle, \\ \langle k_2, x_{2,1}, x_{2,2}, \cdots, x_{2,m_2} \rangle, \\ \vdots \\ \langle k_n, x_{n,1}, x_{n,2}, \cdots, x_{n,m_n} \rangle \end{pmatrix} .$$

Decrypt Phase. *In this phase, the receiver gives the indices (i, j) of the $x_{i,j}$ values that he wishes to receive. The state-based process begins by calculating the initial state $\mathcal{S}_0 \leftarrow InitialState(C)$, and then evolving the state and providing answers to queries as*

$$(t_k, \mathcal{S}_k) \leftarrow Decrypt_{\mathcal{K}_s}(\mathcal{S}_{k-1}, C, i_k, j_k),$$

for $k = 1, 2, \ldots, q$ for some number of queries q. We require that index information be embedded in t_k such that there is a function "ind" that extracts this information as

$$(i_k, j_k) \leftarrow ind(t_k).$$

PostProcess Phase. *This phase takes the results of the Decrypt calls and either fails (giving $\perp$ as the result) or produces q plaintext values as*

$$\langle v_1, v_2, \ldots, v_q \rangle \leftarrow PostProcess(t_1, t_2, \cdots, t_q)$$

3.2   Desired Security Properties

A secure GNIOT scheme must satisfy the following properties:

Correctness. If the Alice and Bob follow the above steps in the prescribed way, and the index set defined by $\mathcal{I} = \{(i, j) \mid ind(t_k)$ for $1 \leq k \leq q\}$ is well-formed, then the values produced by *PostProcess* are exactly the requested plaintext values such that $v_k = x_{ind(t_k)}$ for $k = 1, \ldots, q$.

Sender's Privacy. Bob should not be able to obtain any information about the remaining $m_i - k_i$ elements in each set S_i.

Receiver's Privacy. Alice should not be able to determine which k_i values Bob received from each set.

In a non-interactive process, where there is no communication with the sender in the *Decrypt* or *PostProcess* phases, the Receiver's Privacy property is trivially met. For the Sender's privacy, we define a game played between a probabilistic, polynomial time (PPT) adversary $\mathcal{A}$ and an oracle, where the oracle runs the parts of the parts of the protocol associated with the Sender.

1. The adversary supplies a plaintext input to the GNIOT scheme where each input has two different possibilities:

$$\langle (x^0_{1,1}, x^1_{1,1}), (x^0_{1,2}, x^1_{1,2}) \cdots, (x^0_{1,m_1}, x^1_{1,m_1}) \rangle$$
$$\langle (x^0_{2,1}, x^1_{2,1}), (x^0_{2,2}, x^1_{2,2}) \cdots, (x^0_{2,m_2}, x^1_{2,m_2}) \rangle$$
$$\vdots$$
$$\langle (x^0_{n,1}, x^1_{n,1}), (x^0_{n,2}, x^1_{n,2}) \cdots, (x^0_{n,m_n}, x^1_{n,m_n}) \rangle$$

2. The oracle generates an independent random bit $r_{i,j} \in_R \{0,1\}$ for each pair. The oracle then creates a single GNIOT input by using inputs $x^{r_{i,j}}_{i,j}$ for $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, m_i$ and calls the *Transmit* function. The resulting C is sent back to the adversary.

3. (a) $\mathcal{A}$ makes a series of calls to *Decrypt*, receiving values $t_1, t_2, \ldots, t_q$.
   (b) The adversary is free to perform any computation using the information it obtained, possibly calling the *PostProcess* function of the GNIOT scheme.
   (c) The adversary finally outputs a guess g and an index (a, b).

The adversary wins this game if $g = r_{a,b}$, but we are only interested in when the adversary wins to learn a value that it shouldn't. Therefore, if $\mathcal{I}$ is the index set for the queries made in Step 3a, we define the "advantage" for adversary $\mathcal{A}$ as

$$Adv_{GNIOT,\mathcal{A}} = \left| Pr[g = r_{a,b} | (a, b) \notin \mathcal{I} \text{ or } \mathcal{I} \text{ not well-formed}] - \frac{1}{2} \right|.$$

The security of a GNIOT scheme is defined as the advantage of the best adversary,

$$Adv_{GNIOT} = \max_{\mathcal{A}}(Adv_{GNIOT,\mathcal{A}}),$$

and the scheme satisfies the Sender Privacy property if Adv_{GNIOT} is negligible.

3.3 TPM-Based Solution

Our TPM-based solution makes use of both a standard symmetric cipher and a public key cryptosystem in which use of the private key is count-limited by the TPM. Based on previously defined parameters λ and l_d we define several additional parameters for our solution, as given below.

- l_b (Encrypted Data Length): Length of the data after encryption with the symmetric cipher.
- l_s (Symmetric Key Length): Length of the key for the symmetric cipher. Must be polynomial in λ.
- l_p (Public Key Payload Size): Length of data that can be encrypted with the public key scheme. Must be polynomial in λ, and must satisfy $l_p \geq l_b + l_s$.

The basic idea behind our GNIOT scheme is to doubly encrypt the values $x_{i,j}$ with the symmetric scheme and the public key scheme so that the count-limit restriction ensures that not too many values are decrypted, and a secret sharing scheme is used to make sure that at least k_i are decrypted from each set to allow

recovery of the symmetric key for the final plaintext decryption. As a result, exactly k_i values from each set must be decrypted. Our formal definition follows the phases defined in Section 3.1.

Setup Phase. Bob creates an N-time use count limited key pair [8] (K_p, K_s), where $N = (k_1 + k_2 + \cdots + k_n)$. For further assurance in subsequent key transfer, Bob can certify K_p using an Attestation Identity Key (AIK).

Transmit Phase. The plaintext values $x_{i,j}$ provided to the *Transmit* function will be first protected using a symmetric cipher (such as AES), using a session key R that is generated by selecting n partial keys $R_i \in_R \{0,1\}^{l_s}$ and letting $R = R_1 \oplus R_2 \oplus \cdots \oplus R_n$. Next, for each i we compute m_i shares of each R_i using a threshold-k_i secret sharing scheme, such as the polynomial interpolation based scheme due to Shamir [9], and we denote the shares of R_i by $f_i(j)$, for $j = 1, \ldots, m_i$. By using threshold k_i in the secret sharing scheme, we will be able to compute R_i given any k_i of the $f_i(j)$ values. Using $\mathcal{PKE}_{K_p}$ and $\mathcal{SKE}_R$ to denote the public key and symmetric encryption schemes with keys K_p and R, respectively, we doubly encrypt each $x_{i,j}$ along with a share of R_i to give

$$C_{i,j} = \mathcal{PKE}_{K_p}(\langle \mathcal{SKE}_R(x_{i,j}), f_i(j) \rangle). \tag{1}$$

The collection of ciphertexts $C_{i,j}$, for $i \in \{1, 2, \cdots, n\}$ and $j \in \{1, 2, \cdots, m_i\}$, is then the output of the *Transmit* function.

Decrypt Phase. The only state used in our implementation is in the virtual monotonic counter maintained by the TPM, so all state operations are implicit in the use of count-limited keys. $Decrypt_{\mathcal{K}_s}(\mathcal{S}_{k-1}, C, i_k, j_k)$ then just uses $\mathcal{K}_s$ to decrypt C_{i_k,j_k}, and bundles the resulting values with the index (i_k, j_k) to give

$$t_k = \langle i_k, j_k, \mathcal{SKE}_R(x_{i_k,j_k}), f_{i_k}(j_k) \rangle.$$

PostProcess Phase. For the final *PostProcess* stage, let $\mathcal{I} = \{(i_k, j_k)|1 \leq k \leq q\}$ be the index set of requests made in the Decrypt phase. Then Bob extracts the shares $f_{i_k}(j_k)$ from each t_k, and for each $i \in \{1, \ldots, n\}$ combines the shares corresponding to $\mathcal{I}(i)$ to recover each R_i. These values are then exclusive-ORed together to recover the symmetric key R, which is used to decrypt the plaintexts x_{i_k,j_k}.

3.4   Security Analysis

In this section, we formally prove that our scheme has the required security properties. We use standard security definitions of public key encryption and symmetric key encryption schemes (for example, see [1]).

Theorem 1. *If PKE is an IND-CCA2 secure public key scheme and SKE is a IND-CCA2 secure symmetric cipher, then a probabilistic, polynomial time adversary $\mathcal{A}$ can win the GNIOT game with non-negligible probability if and only if $\mathcal{I}$ is a well-formed index set and $(a, b) \in \mathcal{I}$.*

Proof

Case 0. $(a, b) \in \mathcal{I}$, and $\mathcal{I}$ is a well-formed index set.

It is easy to see that the PPT adversary $\mathcal{A}$ wins in this case: If $\mathcal{I}$ is a well-formed index set, $\mathcal{A}$ can obtain exactly k_i values from set S_i, by calling the decrypt function, which returns $t_{i,j}$ values as the decryption of the corresponding $C_{i,j}$ values in each set. If $(a, b) \in \mathcal{I}$, then $\mathcal{A}$ can call the PostProcess function to correctly obtain corresponding value $x_{a,b}$.

Case 1. $(a, b) \notin \mathcal{I}$, where $\mathcal{I}$ is a well-formed index set.

Let $\mathcal{A}$ be a PPT adversary that wins the GNIOT game with non-negligible probability, i.e. $\mathcal{A}$ distinguishes between the encryptions of $x_{i,j}^0$ and $x_{i,j}^1$ with non-negligible probability. We can use $\mathcal{A}$ to construct a PPT adversary $\mathcal{A}'$ that attacks the CCA security of the PKE as follows: $\mathcal{A}'$ obtains pk from the PKE oracle which it passes along to $\mathcal{A}$, and then receives the values $x_{i,j}^b$ from $\mathcal{A}$, where $i \in \{1, \dots, n\}$, $j \in \{1, \dots, m_i\}$, and $b \in \{0, 1\}$. $\mathcal{A}'$ picks values $R_1, \dots, R_n$ and computes R and the shares $f_i(j)$ of each R_i as in the `GNIOT.Transmit` phase, and selects an index (a, b) at random. For each $(i, j) \neq (a, b)$, $\mathcal{A}'$ picks $r_{i,j}$ at random and computes $C_{i,j}$ according to (1). For index (a, b), $\mathcal{A}'$ submits $\langle \mathcal{SKE}_R(x_{a,b}^0), f_i(j) \rangle$ and $\langle \mathcal{SKE}_R(x_{a,b}^1), f_i(j) \rangle$ to the PKE oracle, which returns the encryption of one of these values, which $\mathcal{A}'$ uses for $C_{a,b}$. $\mathcal{A}'$ the sends all of the $C_{i,j}$ values to $\mathcal{A}$ as the output of `GNIOT.Transmit`.

In the next stage of the GNIOT game, $\mathcal{A}$ requests the decryption of values $C_{i,j}$, and as long as $(i, j) \neq (a, b)$, $\mathcal{A}'$ can answer these directly by providing $x_{i,j}^{r_{i,j}}$. If $\mathcal{A}$ requests the decryption of $C_{a,b}$, then $\mathcal{A}'$ outputs $\perp$, and quits the game. After q queries $\mathcal{A}$ outputs an index (a', b') and a guess g. If $(a', b') = (a, b)$ then $\mathcal{A}'$ outputs g as its own guess in the PKE game, and if $(a', b') \neq (a, b)$, $\mathcal{A}'$ outputs $\perp$ and quits the game.

For $\mathcal{A}'$ to win this game, $\mathcal{A}'$'s randomly chosen index (a, b) must be the same as $\mathcal{A}$'s selected index (a', b') (which occurs with probability $1/N$) and $\mathcal{A}$ must win the GNIOT game. Therefore

$$Pr[\mathcal{A} \ wins] = \frac{1}{N} Pr[\mathcal{A}' \ wins],$$

and so $Pr[\mathcal{A}' \ wins] = N \cdot Pr[\mathcal{A} \ wins] \leq N \cdot Adv_{PKE}$. Since PKE is an IND-CCA2 secure public key scheme, Adv_{PKE} is negligible, and therefore the probability that $\mathcal{A}$ wins the GNIOT game is also negligible (as required for this case).

Case 2. $(a, b) \in \mathcal{I}$ but $\mathcal{I}$ is *not* a well-formed index set.

Let $\mathcal{A}$ be a probabilistic, polynomial time (PPT) adversary that plays the GNIOT game and attacks the TPM-based scheme. The intuition behind this case is that in order for $\mathcal{A}$ to win the GNIOT game in this case, it must either break the SKE scheme to decrypt $\mathcal{SKE}_R(x_{a,b})$ without knowing R, or must break the PKE scheme to gain additional information about R.

Define game $\mathcal{G}_1$ as the GNIOT game as defined in definition 3, i.e., $\mathcal{A}$ tries to distinguish between the encryptions of $x_{i,j}^0$ and $x_{i,j}^1$ for some (i, j). Now

let us define a modified game $\mathcal{G}_2$, where instead of using the real symmetric key R, the transmit oracle (in part 3 of the GNIOT game) uses a different, independent, random key, $\widetilde{R}$, to encrypt the values in each set. Let T_1 be the event that $\mathcal{A}$ wins in game $\mathcal{G}_1$ and T_2 be the event that $\mathcal{A}$ wins in game $\mathcal{G}_2$.

We can use $\mathcal{A}$ to construct a PPT adversary $\mathcal{A}'$ that attacks the CCA security of the PKE scheme. In particular, since $\mathcal{I}$ is not well-formed, there must be some set i such that $|\mathcal{I}(i)| < k_i$, so R_i and hence R is independent of the decrypted shares of R_i. Therefore, unless $\mathcal{A}$ can get some information from the non-decrypted $C_{i,j}$ values it gets no information about R and so must break the SKE scheme.

$\mathcal{A}'$ gets public key K_p from the PKE game. $\mathcal{A}'$ picks random key R and computes all R_i values and shares $f_i(j)$. Next, $\mathcal{A}'$ picks a random index (a', b'), and for all $(i, j) \neq (a, b)$ computes $C_{i,j}$ for random selection $r_{i,j}$ exactly as our GNIOT algorithm. For index (a', b'), $\mathcal{A}'$ substitutes a random share $\widetilde{f}_{a'}(b')$ in place of the real $f_{a'}(b')$ for one alternative:

$$P^0_{a',b'} = \langle \mathcal{SKE}_R(x^0_{a',b'}), f_{a'}(b') \rangle \qquad P^1_{a',b'} = \langle \mathcal{SKE}_R(x^1_{a',b'}), \widetilde{f}_{a'}(b') \rangle \ .$$

These two plaintexts are then passed along to the PKE game as the challenge plaintexts, and we receive a ciphertext $C_{a',b'}$ back, which is the encryption of one of these. Note that if $P^0_{a',b'}$ is chosen, the key used is the correct key constructed from the share $f_{a'}(b')$, so we're perfectly simulating the GNIOT game (game $\mathcal{G}_1$). On the other hand, if $P^1_{a',b'}$ is chosen then the fake share $\widetilde{f}_{a'}(b')$ makes the symmetric key R independent of the key reconstructed from the shares, and so we're perfectly simulating game $\mathcal{G}_2$. Let $\delta \in \{0, 1\}$ represent the choice made by the PKE game.

When $\mathcal{A}$ produces an index (a, b) and guess g, if $(a, b) = (a', b')$ we output "fail" and quit. When $(a, b) \neq (a', b')$, if $g = r_{a,b}$ (i.e., the guess is correct), we output $\widehat{\delta} = 0$ as our guess in the PKE game; otherwise we output $\widehat{\delta} = 1$. Analyzing the probability that output $\widehat{\delta}$ is correct,

$$\begin{aligned}
Pr[\widehat{\delta} = \delta] &= Pr[g = r_{a,b}|\delta = 0]Pr[\delta = 0] + \\
&\qquad\qquad (1 - Pr[g = r_{a,b}|\delta = 1])\, Pr[\delta = 1] \\
&= \frac{1}{2}\, Pr[T_1] + \frac{1}{2}\, (1 - Pr[T_2]) \\
&= \frac{1}{2}\, (Pr[T_1] - Pr[T_2]) + \frac{1}{2} \ .
\end{aligned}$$

Since $\widehat{\delta} = \delta$ means $\mathcal{A}'$ wins the PKE game,

$$Pr[T_1] - Pr[T_2] = 2 \left(Pr[\widehat{\delta} = \delta] - \frac{1}{2} \right) \leq 2\, Adv_{PKE} \ . \tag{2}$$

Next we use $\mathcal{A}$ to construct an adversary $\mathcal{A}''$ playing the standard SKE game. $\mathcal{A}''$ selects R_i values and computes R and the shares $f_i(j)$ as in the

algorithm, and also generates a public keypair (K_p, K_s). $\mathcal{A}''$ initiates the SKE game, which causes the SKE oracle to select a symmetric key that is random and independent of R, and which will be used for all symmetric encryptions that are provided to $\mathcal{A}$ — this means that $\mathcal{A}$ is actually playing game $\mathcal{G}_2$. Next, $\mathcal{A}''$ selects a random index (a', b'), picks a random bit $r_{i,j}$ for each $(i, j) \neq (a', b')$, and uses the SKE encryption oracle to compute plaintexts $P_{i,j} = \langle SKE.Encrypt(x_{i,j}^{r_{i,j}}), f_i(j) \rangle$. $\mathcal{A}''$ then passes both $x_{a',b'}^0$ and $x_{a',b'}^1$ as the challenge plaintexts to the SKE game, and receives a ciphertext c back, which it uses to compute $P_{a',b'} = \langle c, f_{a'}(b') \rangle$. Now $\mathcal{A}''$ uses it's public key K_p to compute $C_{i,j} = \mathcal{PKE}_{K_p}(P_{i,j})$ for all (i, j).

Finally, $\mathcal{A}$ will produce index (a, b) and a guess bit g. If $(a, b) \neq (a', b')$ we output "fail" and quit; otherwise, we pass along the guess g as $\mathcal{A}''$'s guess in the SKE game. $\mathcal{A}''$ wins exactly when it's index (a, b) is correct and when $\mathcal{A}$ wins (in game $\mathcal{G}_2$), so

$$Adv_{SKE,\mathcal{A}''} = \frac{1}{N} Pr[T_2].$$

This means that $Pr[T_2] \leq N \cdot Adv_{SKE}$. Combining with equation (2), we get

$$Pr[T_1] - N \cdot Adv_{SKE} \leq 2 \ Adv_{PKE}$$
$$Pr[T_1] \leq 2 \ Adv_{PKE} + N \cdot Adv_{SKE}$$

Therefore, $Adv_{GNIOT} \leq 2 \ Adv_{PKE} + N \cdot Adv_{SKE}$, and since PKE and SKE allow only negligible advantage, Adv_{GNIOT} is also negligible. ∎

4   Non-interactive Secure Mobile Agents

In this section we give an example application of the GNIOT primitive, in which we significantly improve the efficiency of secure mobile agent protocols. In the mobile agent paradigm, an agent owner, also called the *originator*, creates software agents that can perform tasks on her behalf. After creating the agents for some specific purpose, the originator sends them out to visit various remote hosts, where the agents perform computations on behalf of the originator. When the agents return home, the originator retrieves the results of these computations from the agents. The utility of this paradigm is based on the ability of the originator to go *offline* after sending the agents out, and, ideally, no further interaction between the agent and the originator or the host should be required.

The agent and its state travel to potentially untrusted hosts, where it is at the mercy of the execution environment provided by that host, so the problem of protecting the agent's computation and state from malicious hosts is quite challenging. Secure Function Evaluation (SFE) provides a means to protect these computations, as described more carefully below, but requires interaction between the remote hosts and either the originator or proxies for the originator. Examining this interaction more closely, we will see that the only interaction required is for a set of oblivious transfers, and so by applying our GNIOT implementation we

remove the interaction requirement for secure mobile agent computation. Since the oblivious transfer and the corresponding interaction is a major bottleneck in implementations of these protocols [6], the resulting non-interactive secure agent computations improve the practicality of these techniques significantly.

In the following sections, we review SFE concepts and techniques, explore the relation between SFE and secure mobile agent computation, and outline an improved agent protocol using the GNIOT primitive from the previous section.

4.1   Secure Function Evaluation

Two-party Secure Function Evaluation (SFE) is a cryptographic primitive that allows two parties, Alice and Bob (with inputs a and b respectively) to compute a function $(A, B) \leftarrow f(a, b)$ such that Alice learns output value A and Bob learns output B, and neither party learns anything more than what follows from its own values. Yao showed that for any polynomial-time computable function f, there exists a polynomial time SFE protocol [15]. The function is represented as an *encrypted circuit* where the values on the input wires are random strings (called signals) instead of the actual boolean values, and the mapping of the random signals to the real inputs is kept secret. Through carefully-specified truth tables that allow evaluation of gates without needing to know the semantics of the random signals, the encrypted circuit can be evaluated without any information being revealed to the evaluator. The result of the evaluation is in encoded form as well, and to decode the output, knowledge of the mapping of the random signals to the real outputs is required.

In this two-party protocol, Alice creates an encrypted circuit to evaluate the desired function. Then Alice sends the encrypted circuit (along with a proof that the circuit was constructed properly if Alice isn't trusted) along with the random signals corresponding to her input to Bob. She also sends a mapping which will allow Bob to decode his output (B) at the end of the computation. Bob must somehow learn the random signals for his input b, but he cannot be given the full input-to-signal mapping. To accomplish this, he engages in a 1-out-of-2 oblivious transfer protocol with Alice for *each bit* of his input, after which Bob knows the signals for his input bits while Alice learns nothing about which signals Bob received (i.e., Bob's input b). Bob now evaluates the encrypted circuit, having obtained random signals corresponding to both inputs a and b, and returns the resulting encrypted form of Alice's output A to her, which she can decode. Bob uses the previously-supplied mapping for his output signals to decrypt his output. Note that the only interaction required between Bob receiving the circuit and evaluating the circuit is the set of 1-out-of-2 OTs that he uses to receive the random signals for his input, and the form of this operation is exactly an instance of our GNIOT primitive.

4.2   Application of SFE to Mobile Agents

When an agent visits a host, it carries with it some state from previous computations, and performs a computation using this state and some input from the

host being visited. Output of this computation consists of a new agent state, and possibly some output provided to the host. The agent state (both old and new) are "owned" by the agent, and should be protected from potentially malicious hosts, whereas the host input and output are "owned" by the host and should likewise be protected from potentially malicious agents. For the sake of efficiency, we also allow a host or the agent to provide some non-sensitive, unprotected data to the computation. We refer to this as the "Agent Data", and as a result we formalize an agent computation as the 3-input, 2-output computation illustrated in Figure 1.
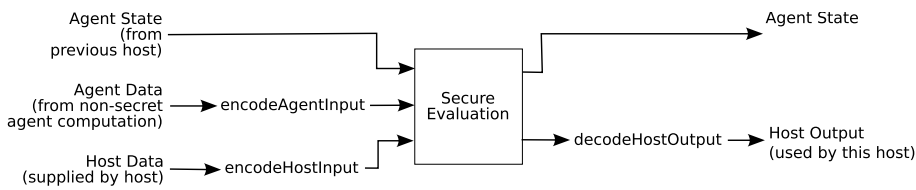

Fig. 1. Agent Computation at a Remote Host

In order to secure this computation we can use two-party Secure Function Evaluation, where one party (the originator) controls the top input and output in the figure, and the other party (the host) controls the bottom two inputs and the bottom output in the figure. Unfortunately, the standard SFE technique described in the previous section requires interaction between the parties, meaning the originator could not be offline, violating a basic property of mobile agent computation. Two existing solutions to the secure agent problem get around this in different ways: a protocol due to Algesheimer *et al.* [2] uses a trusted third party as a proxy for the originator in the oblivious transfer, and a protocol due to Tate and Xu [11,14] (the "TX protocol") uses threshold cryptography and collections of other agents to stand in for the originator. As noted in the previous section, the required oblivious transfer (a 1-out-of-2 transfer for each bit of the host's input) is exactly an instance of GNIOT, and by using our TPM-based implementation we can completely remove any need for interaction in the agent computation. Due to the similarity with the TX protocol, we call this new protocol the "GTX protocol."

4.3   The GTX Protocol

In this section we describe all of the steps required by our non-interactive secure agent protocol. We break down the required operations into three phases, initialization, evaluation, and finalization, corresponding to the three phases of the SAgent software framework for secure mobile agents [5]. While all steps are described here, space limits preclude a detailed descriptions and readers unfamiliar with previous work in secure agents may want to refer to earlier papers in this area [2,11,14].

1. *Initialization:* The originator creates an *encrypted circuit* for each sensitive computation to be carried out at a host — the square box in Figure 1. As outlined in section 4.1, encrypted circuits are special boolean circuits where the signals on the wires are random strings instead of 0 or 1. Since the encrypted circuit can be evaluated with encoded signals, the agent state and inputs must be encoded and incorporated into the agent.

   For the GTX protocol, the participating hosts are assumed to have TPMs, with unambiguous identities which can be verified by an agent originator. Each host willing to accept agents and supply n-bit inputs executes the *Setup* phase of GNIOT to generate n-time use keys that are made available to users wishing to send agents. When an originator wants to send out agents, the originator executes the *Transmit* phase of the TPM-based GNIOT scheme, where $m_i = 2$ and $k_i = 1$ for all $i \in \{1, \ldots, n\}$, and we let $x_{i,1}$ and $x_{i,2}$ be the two signals corresponding to boolean values 0 and 1 for host input bit i. Note that the output of the *Transmit* phase of GNIOT is exactly what the hosts will need to decrypt exactly one random signal for each of its n input bits. In creating the agent, the originator bundles together the encrypted circuit, the output C of the GNIOT *Transmit* phase, and the host's output-to-boolean mapping and includes all of this information in the agent. The originator keeps the final state signal-to-boolean mapping for use in decrypting the final agent state when it returns after having visited the hosts.

2. *Evaluation:* In the evaluation phase, the host has received an agent, which carries with it the values described above. If the host's input is made up of bits $\langle b_1, b_2, \ldots, b_n \rangle$, the host calls the GNIOT.*Decrypt* with indices $(i, b_i + 1)$ for $i = 1, \ldots, n$. Running *PostProcess* on the results of these *Decrypt* calls will provide $\langle x_{1,b_1+1}, x_{2,b_2+1}, \ldots, x_{n,b_n+1} \rangle$, which are exactly the random signals needed to evaluate the encrypted circuit. Note that if the host tries to cheat either by requesting both signals corresponding to a single input bit or by requesting more than the allowed number of decryptions, the GNIOT protocol guarantees that the host learns nothing at all about the random signals used by this encrypted circuit. After evaluation of the encrypted circuit, the host uses the output signal-to-boolean mapping supplied by the originator (and carried by the agent) in order to decrypt its input.

3. *Finalization:* When the agent returns to the originator, its final state will be decrypted by the originator.

5   Conclusion

In this paper, we have shown how to remove interaction requirements in the fundamental cryptographic primitive of oblivious transfer to create an expanded cryptographic primitive called "generalized non-interactive oblivious transfer" (GNIOT). Based on recent research which shows how to instantiate count-limited objects using the monotonic counter in trusted platform modules, we outline how to use count-limited objects to efficiently instantiate an oblivious transfer primitive while removing the interaction requirements necessary in such a protocol.

We provide rigorous proofs that under an assumption of secure TPMs (and standard complexity assumptions), our construction provides the same security properties as those of standard oblivious transfer. In addition, we show how to apply the GNIOT primitive to develop a secure mobile agent protocol (called the GTX protocol) where strong security guarantees can be achieved without the interaction requirements necessary in previous secure agent protocols.

References

1. Abe, M., Gennaro, R., Kurosawa, K., Shoup, V.: Tag-KEM/DEM:A New Framework for Hybrid Encryption and A New Analysis of Kurosawa-Desmedt KEM. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 128–146. Springer, Heidelberg (2005)
2. Algesheimer, J., Cachin, C., Camenisch, J., Karjoth, G.: Cryptographic security for mobile code. In: Proc. of the IEEE Symposium on Security and Privacy, pp. 2–11 (2001)
3. Bellare, M., Micali, S.: Non-interactive oblivious transfer and applications. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 547–557. Springer, Heidelberg (1990)
4. Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. SIAM J. Comput. 33(1), 167–226 (2003)
5. Gunupudi, V., Tate, S.R.: SAgent: A Security Framework for JADE. In: AAMAS 2006: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, pp. 1116–1118 (2006)
6. Gunupudi, V., Tate, S.R., Xu, K.: Experimental evaluation of security protocols in SAgent. In: Proceedings of the International Workshop on Privacy and Security in Agent-based Collaborative Environments (PSACE), pp. 60–74 (2006)
7. Rabin, M.O.: How to exchange secrets by oblivious transfer. Tech. Rep. TR-81, Harvard University (1981)
8. Sarmenta, L.F.G., van Dijk, M., O'Donnell, C.W., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In: STC 2006: Proceedings of the First ACM Workshop on Scalable Trusted Computing, pp. 27–42 (2006)
9. Shamir, A.: How to share a secret. Communications of the ACM 22, 11 (1979)
10. Strasser, M., Sevnic, P.E.: A software-based TPM emulator for Linux. Master's thesis, Eidgenossische Technische Hochschule (ETH), Zurich, Project web page (2005), `http://developer.berlios.de/projects/tpm-emulator/`
11. Tate, S.R., Xu, K.: Mobile agent security through multi-agent cryptographic protocols. In: Proc. of the 4th International Conference on Internet Computing (IC), pp. 462–468 (2003)
12. Trusted Computing Group, `http://www.trustedcomputinggroup.org`
13. Trusted Computing Group. TPM main specification, version 1.2, revision 103, parts 1–3 (2007), `http://www.trustedcomputinggroup.org`
14. Xu, K., Tate, S.R.: Universally composable secure mobile agent computation. In: Zhang, K., Zheng, Y. (eds.) ISC 2004. LNCS, vol. 3225, pp. 304–317. Springer, Heidelberg (2004)
15. Yao, A.: How to generate and exchange secrets. In: Proc. of the 27th IEEE Symposium on Foundations of Computer Science(FOCS), pp. 162–167 (1986)