BlindTM – A Turing Machine System for Secure Function Evaluation

Rafael Dutra
rtd@berkeley.edu

Benjamin Mehne
bmehne@berkeley.edu

Jay Patel
patel.jay@berkeley.edu

ABSTRACT

Secure Function Evaluation (SFE) is the problem of performing an algorithm while keeping the input, output and intermediate values hidden from curious parties. This problem is of increasing interest due to the prevalence of cloud services that allow users to store data as well as execute code remotely.

Previous implementations of SFE systems use "garbled circuits" in which computation is modeled as logic gates which are evaluated while keeping the truth values of each wire hidden. In this work, we present BlindTM, the first implementation of a "Blind Turing Machine" which performs secure function evaluation by computing on and manipulating the tape of a Turing Machine in an encrypted manner. BlindTM supports several tunable parameters for performance and security, and allows secure computation to be expressed in a more flexible manner than garbled circuits, through states and symbols manipulated by a Turing Machine.

1. INTRODUCTION

Secure Function Evaluation (SFE) is the problem of performing an algorithm (function) securely – keeping the input, output and intermediate values hidden from curious parties. Multiple means exist to implement SFE, including garbled circuits and – more recently – fully homomorphic encryption. Systems that implement SFE are of increasing interest due to the prevalence of cloud services that allow users to store data as well as execute code in the cloud. In this work, we focus on applications that involve two parties – one of which (the client) is computationally bounded. As cloud computation platforms become more common – as platform as a service (PAAS) becomes a more common business model – the ability for a computation to be performed remotely without leaking private data on a cloud server will become of increasing relevance.

For example, a hospital may want to perform some analysis on its patient data that requires a large amount of processing and memory resources. It would be very convenient to have this analysis executed by a cloud service provider, but it is often necessary by law to maintain the confidentiality of the patient data and the outputs of the analysis. Furthermore, if the hospital could have this computation performed with minimal interaction with the cloud server, then it could use its own computation resources for its own needs and contract out for more resources only when necessary. This could be achieved by a system that provides SFE between two parties, with bounded interactions between them.

In this project we designed, implemented, and evaluated the first Blind Turing Machine implementation that allows Secure Function Evaluation to be performed by a curious cloud server. The Blind Turing Machine shown here is based on the construction proposed by Rass [12] to use group homomorphic encryption to implement the transitions of the Turing Machine in a secure manner. In the model we are targeting, the server receives the encrypted algorithm in the form of a Blind Turing Machine, along with the client's encrypted inputs. It then executes the algorithm on the encrypted input, returning an encrypted form of the final output that can be decrypted by the client. The desired security guarantee is that the server cannot derive any information about the data being processed with a non-negligible probability (semantic security). We evaluate our implementation with varying degrees of security guarantees as well as compare with garbled-circuits based implementations.

2. ALTERNATIVE METHODS FOR SECURE FUNCTION EVALUATION

The problem of SFE has been studied under various approaches. The most common technique is the use of Yao's garbled circuits [13, 14]. In Yao's garbled circuits, the idea is to represent an algorithm by a circuit with logic gates and evaluate the circuit while keeping the truth values of each wire hidden. This is a protocol performed by two parties: the Garbler and the Evaluator. The Garbler creates two random keys for each wire (one for each truth value) and sends a garbled truth table of each gate to the Evaluator. The Evaluator can then execute each logic gate on encrypted values.

Other approaches are the ones studied under the more general area of Secure Multiparty Computation (SMC). The main goal of SMC is to allow the evaluation of a function that takes inputs from two or more different parties. Garbled circuits can be used for this purpose: the garbled circuit evaluator can choose some inputs to the logic gates and obtain the corresponding key from the Garbler by using an oblivious transfer. This mechanism is a superset of two-party SFE, since it also allows the evaluation of functions that receive inputs from different parties.

Most recently, in 2009, Fully Homomorphic Encryption (FHE) was shown to be a practical approach to SFE [2]. This method relies on an encryption scheme that is homo-

morphic with respect to the two group operations $(+, \cdot)$. That is, addition or multiplication operations performed on the ciphertext produces a result which, when decrypted, matches the result of applying the identical operations on the plaintext. One disadvantage of this method is that currently known implementations of FHE systems are not efficient when compared to other cryptographic schemes. This mechanism provides an advantage over garbled circuits in that it can represent more easily traditional programs.

3. STATE OF THE ART

Fairplay [8] was the first system which demonstrated the feasibility of two-party secure computation using garbled circuits. Fairplay programs are implemented in a domain-specific language called Secure Function Definition Language. This language is then processed into a circuit representation (in the form of Secure Hardware Definition Language), which is then garbled. Optimizations are applied to eliminate dead code and optimize the gate layout. The resulting garbled circuit is in the form of a Java object which can then be executed on an input. The circuit specification language fixes the size of each input and intermediate variable to enable direct creation of the circuits.

The original Fairplay project was used as the inspiration for many subsequent projects. One such system called TASTY [4], utilized a combination of homomorphic encryption and garbled circuits to improve on the Fairplay implementation and automatically generate protocols for SFE communication. A secure computation framework from [5] is a system that attempts to improve the efficiency of garbled circuit computation over similar Fairplay-based systems. We refer to this framework as FastGC in the comparison section.

More recently, Malka et al. described VMCrypt [7], a software architecture for parallel garbled circuit evaluation with the explicit goals of scalability and software modularity in mind. Their system prominently featured an API (Abstract Programming Interface) to enable developers to integrate garbled circuits into their own applications and to easily create new/reuse circuits as needed.

All of the freely and publicly available implementations of SFE focus on garbled circuits-based systems. This paper is the first implementation of a Turing Machine-based SFE system. In section 6, we provide comparisons against two garbled circuits based systems.

4. CRYPTOGRAPHIC BACKGROUND

In 2013, two constructions for a Turing Machine which computes over encrypted data were introduced. Goldwasser et al. propose a model which uses fully homomorphic encryption [3]. Rass's construction, which is used by the BlindTM, uses a weaker cryptographic primitive, homomorphic public-key encryption with equality test (HPKEET) to build a Turing Machine that works over an encrypted tape, using the homomorphic properties of the encryption. One advantage of this construction is that it does not require a fully homomorphic encryption, which is usually less efficient than other cryptographic primitives.

4.1 Turing Machine architecture

In this project we worked with multi-tape Turing Machines. The advantage of having many tapes is that it provides an easy way to represent programming constructs. Additional tapes may be used to copy arrays of symbols, to implement queues, stacks and other data structures.

Each tape extends infinitely in both directions. Each tape has its own head and all heads can move independently. The machine can read the symbol below the head of any, or all of, its tapes in order to decide the next transition. The machine also maintains a state and uses a finite set of logical rules to determine the next state, the symbols that should be written on each tape, and the movements of each head.

Thus the transitions of the Turing Machine can be represented as a mapping

$$(State, [Symbol]) \to (State, [Symbol], [Move]).$$

For example, the following entry in the mapping

$$(q, [\sigma_1, \sigma_2]) \to (q', [\sigma_1', \sigma_2'], [move_1, move_2])$$

determines that this 2-tape Turing Machine, when in state q and finding symbols σ_1, σ_2 under its heads, should go to state q', replace these symbols by σ_1', σ_2' and move each head according to $move_1, move_2$.

The basic idea of the Blind Turing Machine is that each transition should be transformed into

$$\begin{aligned}(g^q, [g^{\sigma_1}, g^{\sigma_2}]) \to (&Enc_{pk}(q' - q),\\ &[Enc_{pk}(\sigma_1' - \sigma_1), Enc_{pk}(\sigma_2' - \sigma_2)],\\ &[move_1, move_2]).\end{aligned}$$

Some further explanation is required here. Firstly, g is a generator in a group where the discrete logarithm is assumed to be computationally hard. That is, for each value x, we assume that it is infeasible to recover x from g^x. In this way, the exponentiation $x \mapsto g^x$ works as a commitment of the value x.

So the evaluator has access to the commitments of the current state q and the symbols σ_1, σ_2. By using these commitments, the evaluator finds the appropriate transition in the transition table and is able to retrieve encryptions of the differences $\Delta q = q' - q$, $\Delta\sigma_i = \sigma_i' - \sigma_i$.

The HPKEET encryption scheme used here is described in §4.2. It is homomorphic, which allows the evaluator to obtain an encryption of the next state by calculating $Enc_{pk}(q') = Enc_{pk}(q' - q) \cdot Enc_{pk}(q)$. The same procedure is performed on tape symbols σ_i. This scheme also allows equality tests of ciphertexts. It does so by enabling the evaluator (who knows a secret token that allows comparison) to obtain commitments of the plaintexts (that can be compared for equality). In this way, the evaluator can obtain a commitment $g^{q'}$ of the next state. The same can be done for the next tape symbols read from the tapes (which could be in new positions since the heads can move). From these commitments, the evaluator proceeds to the next iteration.

The use of homomorphism for calculating the next configuration was chosen by Rass in its Blind Turing Machine in

order to provide stronger security guarantees against adversaries which are not authorized to run the Turing Machine (those that do not have the authorization token). This design allows, for example, clients to send an Encrypted Turing Machine to a cloud service provider but only send the authorization token at a later date. While the service provider has not received the authorization token, it is not capable of executing the Turing Machine or extracting information from it.

4.2 Homomorphic Public Key Encryption with Equality Test

In order for this construction to work, a Homomorphic Public Key Encryption with Equality Test (HPKEET) must be constructed. We follow the construction proven to be CCA1-secure (chosen-ciphertext attack) in [12]. The HPKEET scheme is described by five procedures $(KeyGen, Enc, Dec, Aut, Com)$. Besides the standard functions for key generation, encryption and decryption, there is an authorization function Aut that outputs a token and a comparison function Com that uses the token to compare two ciphertexts.

The construction is based on an underlying Homomorphic Public Key Encryption (HPKE) scheme which will be denoted by (G, E, D). We assume the ciphertext space in this HPKE primitive is a group $\mathbb{G}$ for which the discrete log problem is hard. Let g, h be two different generators for $\mathbb{G}$. The construction is defined as follows.

KeyGen. let (sk_1, pk_1) and (sk_2, pk_2) be two pairs of keys generated by G. Output $sk = (sk_1, sk_2)$, $pk = (pk_1, pk_2)$.

Enc. sample a random value s. Output the ciphertext $(c_1, c_2, c_3) = (E_{pk_1}(m), g^m \cdot h^s, E_{pk_2}(s))$.

Dec. verify if the ciphertext (c_1, c_2, c_3) is well-formed by checking $g^{Dec_{sk_1}(c_1)} \cdot h^{Dec_{sk_2}(c_3)} = c_2$. In the affirmative case, output $Dec_{sk_1}(c_1)$.

Aut. output sk_2.

Com. to compare ciphertexts $(c_1, c_2, c_3), (c_1', c_2', c_3')$, check if $c_2 \cdot h^{-D_{sk_2}(c_3)} = c_2' \cdot h^{-D_{sk_2}(c_3')}$. In the affirmative case, output 'equal', otherwise 'not equal'.

It is easy to see that this HPKEET scheme is homomorphic if we assume the underlying HPKE scheme is additively homomorphic.

4.3 Damgård ElGamal encryption

The underlying HPKE scheme suggested in [12] is Damgård's version of ElGamal encryption [1]. This is a simple modification to the original ElGamal encryption that allows it to be proved CCA1-secure [6]. In our implementation, we use the group of invertible integers modulo p for a prime number p. The number of bits of p is a configurable security parameter.

In the original ElGamal encryption, the secret key is a number x and the public key is the commitment g^x. We assume the discrete logarithm is hard for group $\mathbb{G}$, so that obtaining x from g^x is infeasible. Then to encrypt a message m, a random number r is sampled and the ciphertext is constructed as $(g^r, m \cdot (g^x)^r)$.

In Damgård's version, an extra term is included in the ciphertext. The secret key now consists of two different numbers x, y and the public key is composed by the commitments g^x, g^y. Then the ciphertext is constructed as $E_{pk}(m) = (g^r, m \cdot (g^x)^r, (g^y)^r)$ for a random r. With this extra term, it is possible to check if the ciphertext is valid before decrypting it. This is enough to prove the system CCA1-secure.

The problem of using Damgård ElGamal encryption (DEG) as the HPKE scheme used to construct HPKEET is that DEG is multiplicatively homomorphic. It is easy to see that $E_{pk}(m) \cdot E_{pk}(m') = E_{pk}(m \cdot m')$. To turn this scheme into an additively homomorphic one, we can just encrypt a commitment v^m of the plaintext m, instead of encrypting m directly. Here, v may be any generator of the group $\mathbb{G}$.

In particular, the HPKEET ciphertext is adapted to be $(c_1, c_2, c_3) = (E_{pk_1}(g^m), g^m \cdot h^s, E_{pk_2}(h^s))$. This way we can still allow comparisons in the presence of the token sk_2, since this token can be used to decrypt h^s and then obtain the value of g^m.

The problem now is how to allow decryption, since the plaintext m is 'committed' as g^m. This can be solved if we assume all plaintexts m come from a reasonably small space. Then we can maintain a lookup table that maps commitments to plaintexts.

Note that this extra step of encrypting commitments v^m instead of directly encrypting m is only necessary because the chosen HPKE in our implementation was DEG, which is multiplicatively homomorphic. If we used a HPKE scheme which is already additively homomorphic, such as Paillier encryption [10], no modification would be needed. Paillier encryption will be explored in a future work.

4.4 Security challenges

The method described above for creating an encrypted Turing Machine that operates over encrypted data is not by itself enough to achieve security. There are some challenges that need to be addressed to allow this Blind Turing Machine to be applied in a real secure system.

Firstly, the head movements are completely unencrypted to the evaluator. As will be discussed in §5.2, we should make sure the Turing Machine has oblivious head movements to avoid leaking information about the inputs, and the algorithm being performed on the input.

Another problem is that the evaluator has access to the comparison token and therefore is able to compare any two tape symbols or any two states. This may leak significant information about the contents of the inputs and the behavior of the Turing Machine. For example, the evaluator can readily calculate the frequencies of different symbols in the input tape.

We address this problem by performing a blow up of state

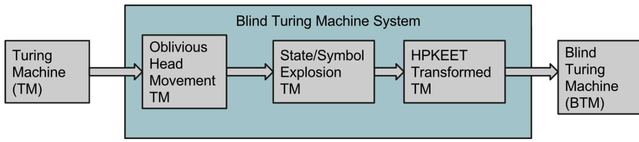


Figure 1: Overview of the system: Everything within the shaded blue rectangle is performed by the BlindTM system.

Figure 2: Overview of how the system works: The BTM (BlindTM) is produced as from Figure 1, the client encrypts the input tape as described in §4.2, the server runs the BlindTM, and the client decrypts the output as in the same section.

and symbol spaces, as will be described in §5.3. In [12], each state and tape symbol is mapped into one random plaintext. The plaintext space is assumed to be much larger than the state space and symbol space used by the Turing Machine. This is required to guarantee a high min-entropy and avoid simple brute-force attacks. In our implementation, each state and symbol can actually be mapped to more than one plaintext, in order to prevent attacks based on ciphertext comparisons.

5. BLINDTM IMPLEMENTATION

BlindTM is written in the Java language. The system implemented takes a Turing Machine and transforms it into a Blind Turing Machine through a series of three phases. We also include infrastructure for running an unmodified Turing Machine, specified with our API. The application of the phases can be seen in Figure 1. The resulting Blind Turing Machine, though being Turing complete, requires further infrastructure than a regular Turing Machine specified in our API. The Blind Turing Machine receives encrypted tape values as its input and outputs an encrypted form of the tape after the computation has completed. This design can be seen in Figure 2.

5.1 Turing Machine Representation

There are many representations of Turing Machines available, and many of which have limitations that are unnecessary for the proofs used in [12]. In order to construct the most useful Turing Machine representation, as many limitations as possible were removed while maintaining the necessary components of the Turing Machine. In our representation, a Turing Machine can have any number of tapes, any size alphabet (specified in bits) for the tape cells, any number of states, and any number of deterministic transitions up to the limits imposed by the Java Virtual Machine environment. Furthermore, we do not require that a state have a complete transition table. During the execution of a Turing Machine as specified in our implementation, if a state is reached such that the value read from the tape has no transition, it is taken to be a transition to a halting statement. In general, we expect that such transitions will represent an error in logic in the application. Future work may use the final state of the Blind Turing Machine execution to check

```
qCompl, 0, qCompl, 1, >
qCompl, 1, qCompl, 0, >
```

Figure 3: Excerpt of a simple binary complement program as input for BlindTM. The first transition can be read as "when in state `qCompl` and the head is on `0`, transition to state `qCompl`, write a `1` to the tape, and move the tape right."

whether the program reached a state that the programmer specified to be a halting state.

In our system, the programmer constructs a TuringMachine object which must specify the number of tapes, the alphabet bit size, starting/initial state, and the initial value for each tape cell. This TuringMachine then must have its transition table loaded. Transitions are specified by a mapping $(State, [Symbol]) \rightarrow (State, [Symbol], [Move])$. Users can input the transitions in an easy to understand text format shown in Figure 3. Furthermore, we have a code generation interface which uses lambda functions to more efficiently generate transition tables for programs with a large alphabet space and many states.

The system maps each state identifier to a distinct integer. A tape cell value is read from each tape and is used as a lookup table to find the next state. Three head movements are possible for each tape: left, right or no movement. The inputs of the Turing Machine can be loaded into one or more tapes.

Once the TuringMachine object is constructed, it is passed to the three phases of the BlindTM system.

5.2 Oblivious Turing Machine

The first phase of the BlindTM system is the transformation that makes it oblivious with regards to head movements. Consider the Turing Machine that reads the value at the initial head position on a binary tape, and moves left if it is 1 and right if it is 0. By running this Turing Machine, one can clearly obtain information about the input from its head movement. Furthermore, the head movement in this example is not necessary for correctness (the output tape is unchanged, and one halting state can be used).

From this example, it is clear that the head movement may reveal characteristics of the input. In order to mitigate this issue, we create an Oblivious Turing Machine. An Oblivious Turing Machine is one where the position of the head is determined by the size (and not content) of the input and the time step. For instance, the typically implemented Turing Machine that bitwise AND's two binary tapes and prints to an output tape is oblivious – the Turing Machine starts on the left side of the input and moves right, writing the output to the output tape at each step. The head movement in this case is only a function of the input size and the time – any content of the same size yields the same head movements.

Programmers using the BlindTM API must specify if their Turing Machine is already oblivious or our system will attempt to change them into an Oblivious Turing Machine. Many Turing Machines – machines adding two numbers, performing bitwise operations, calculating Hamming distance –

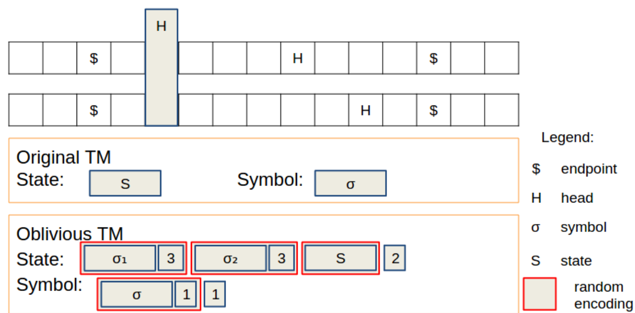


Figure 4: Transformation used to obtain an Oblivious Turing Machine.

are typically implemented in an Oblivious Turing Machine or can be easily transformed into one. Performing these changes manually prevents an increase in overhead by the automatic transformation. It should be noted that detecting whether a Turing Machine is already oblivious is non-trivial and, if completeness is desired, is reducible to the Halting Problem. Naively, one may think that if each state has the same movement for all transitions, then the Turing Machine is oblivious, but this is not true. Consider a state that always has the head move left, but half of its transitions lead to a state that always moves right and the other half lead to a state that always moves left. If only one movement type is used for all except the halting state, then it is still not necessarily oblivious, as two inputs of the same size may have different length execution paths. For a Turing Machine to be oblivious, all paths must have the same movement pattern, which can only be detected easily in a narrow set of instances. Exploring the space of heuristics to automatically detect Oblivious Turing Machines in general is left to a future work.

5.2.1 Oblivious transformation

Figure 4 represents the method we used to transform the original Turing Machine into an oblivious one. Special markers (the symbols $ in the figure) are used to represent left and right bounds of the tapes. Intuitively, they represent the bounds of where the heads of the original Turing Machine could have been at the current stage being simulated. These heads are represented by the letters H in the figure and are always between the two end markers. The new heads of the transformed (oblivious) Turing Machine always move together (vertical bar in figure) and will be referred to as the superhead. At each stage, the superhead moves from left to right and from right to left, pushing forwards each of the end markers $ by one unit.

This Oblivious TM is able to simulate the original TM by manipulating the heads of the original TM when the superhead goes over them. Local buffers are used to store the required data for these manipulations. They are stored as part of the state of the new TM, as shown in Figure 4. At the beginning of each stage, there is one symbol buffer (σ bits) per tape that stores the symbol below each head and one state buffer (S bits) that stores the current state of the original TM. Using these values, the Oblivious TM is able to determine the next transition: next symbols to write under the heads, next movements to perform and next state to go to. The state is stored in the state buffer of S bits and each new symbol to be written is stored in a symbol buffer of σ bits. The movements occupy two bits each and are stored as part of the three control bits attached to each symbol buffer.

After this first step, the symbol buffers contain the symbols that should be written under each head. During the pass of the superhead, those symbols are written at the positions of the original heads (marked H) and the heads are moved if necessary. The new symbol below the head is read and stored in the buffer. If the original head is supposed to move left, this is done during the left pass of the superhead. If it needs to move right, it is done during the right pass. If the head needs to stay stationary, this can be done in any of the passes (left, for example).

In this construction, the state of the Oblivious TM with T tapes requires $T(\sigma+3)+S+2$ bits and each tape cell requires $\sigma+2$ bits, including the control bits that determine the movements of the heads and the places to read and write the special markers H and $ in the tapes. A naive implementation of the transition function would be a lookup table with $2^{T(2\sigma+5)+S+2}$ entries. To reduce the memory overhead, we used two tables. The Transition Table is used at the beginning of each stage to determine the next transition. It does not depend on the symbols currently under the superhead, which are all blank, since the superhead is at the leftmost position of the tapes, which was never used. From this observation, the size of this table can be reduced to $2^{T(\sigma+3)+S}$. The other table is the Symbol Table, used to manipulate the symbol buffers and the symbols of the tapes. In our implementation, only one symbol table can be used for all tapes, allowing its size to be $2^{2\sigma+4}$. For example, in a TM with 3 tapes, 2 bits per symbol and 4 bits per state, the naive implementation would need a table of size 2^{33}, while our implementation uses one table of size 2^{19} and another of size 2^{8}, making the system more practical.

The local buffers are encoded using random values. In Figure 4, each red rectangle represents a block of bits that needs to be encoded. As discussed in §5.3, each of those blocks can be encoded to many different values in order to hide information. The control bits that represent the markers $ and the movement of the superhead do not need to be encoded, because the information they store is always visible to the evaluator of the Turing Machine.

The security of this model is based on the assumption that no information can be extracted from the blocks of bits after they have been encoded as (possibly many) random values. So after the first step that uses the Transition Table, the new symbols that should be written under the heads remain hidden and the new state of the original TM also remains hidden. And since the moves of the heads are encoded in conjunction with the symbol buffers, they also cannot be discovered by the evaluator. Then, during the right pass and the left pass of the superhead, the Symbol Table is used to update the symbol buffers and the symbols below the superhead. At each step, both of these blocks of randomly encoded symbols are updated for each tape, and there are many different representations for each of them. So the evaluator cannot find out where the actual head movements took place. The control bits that indicate where the head

is, which movement the head should perform and where the head needs to be dropped are encoded jointly with the symbol values. So, from the evaluator's perspective, all tape operations look the same, independently of the fact that a given head is being manipulated or not.

This simple construction takes time $O(n^2)$ to simulate n steps of the original TM. There is a construction from Pippenger and Fischer [11] with runtime $O(n \log(n))$, which is the best possible, but this construction is applied to a single-tape TM. We considered implementing this construction after transforming the multi-tape TM to a single-tape one. The transformation from multi-tape to single-tape is very similar to our implementation of the Oblivious TM described above: the single head needs to "find" all the heads at each tape before it can reach a decision for the current stage. So this transformation to a single-tape TM would already produce a $O(n^2)$ overhead, invalidating the possible gain that would be obtained by the $O(n \log(n))$ algorithm of [11].

5.3 State and Symbol explosion

The second phase of the BlindTM system is the blow up of the number of different states and symbols. Before this phase the head movements of the Turing Machine are already obfuscated from any curious observer, but the values of the states and tape symbols are clearly visible. At the end of this phase, a direct comparison between the states/tape cells and the original Turing Machine becomes infeasible. This phase is broken into three parts to counter three issues: the comparison of states related to some known input, the comparison of the tape to some known input, and the comparison of tape writing behavior to that observed with some input.

5.3.1 State-space explosion

The issue of a direct state comparison can be naively handled via duplicating states - for any state q, create some states q', q'', q''', et cetera. For each of these new states, populate them with the same transition functions as q. Then, to make certain that they are attainable in the Turing Machine, randomly select transitions that lead to q and redirect them to one of the $q', q'', q''', \cdots$. Increasing the number of equivalent states does not increase the runtime of the Turing Machine, since each additional state performs the same as the original state and no execution path is increased. An alternative mechanism that could be used to increase the number of states would be to allow for new intermediate states between original states, but this would increase the runtime by some constant factor. The simple state-space explosion method we use here was chosen to minimize the runtime overhead, at the expense of increasing the size of the representation of the Turing Machine. Note that even with the increased number of states, it is possible to revert to the original Turing Machine by deduplicating states that have similar transitions and write the same value to the tape. This can be done inductively, starting at the halting state(s) to recover the entire machine – increasing the state-space is not sufficient to prevent a curious attacker from recovering the state space, which can then be used to determine characteristics of the input. In our implementation, each state is assigned a random identifier and a mapping is kept, emulating a one-time pad. This exposes another attack available to a curious adversary – if the adversary has some auxiliary knowledge of the inputs (or can create some inputs), it can then gain information about how they are processed and thereby gain information about subsequent inputs. This is handled in the third phase of our system. The number of states created from the original state is a controllable security parameter.

5.3.2 Tape-space explosion

In order to mitigate the ease by which an attacker can deduplicate states after the state-space explosion phase, we implement a tape-space explosion. First we create a second, larger alphabet for the tape that is a superset of the original alphabet. Then we map each element on the original tape to some non-empty set of elements in the second tape's alphabet. This is performed for each tape. For each transition that is performed for some tape symbol σ from the original tape, one of the $\sigma', \sigma'', \sigma''', \cdots$ tape symbols assigned to it from the second tape that is used to replace σ. After this transformation, deduplicating the states requires the tape symbols to be first deduplicated so that it is possible to find equivalent transitions. Since the final tape is passed back as the output of the Turing Machine, it is possible to construct a machine via this explosion scheme that is the same as a second machine without this explosion mechanism. The results of these two machines would not be meaningfully different. Note that this explosion also functions like a one-time pad and has the same drawbacks – if the adversary could influence or know from some external source, multiple equivalent inputs, it could use the runtime characteristics to deduplicate the tape space. The number of inputs necessary for an effective deduplication scales with the size of the second tape – this is a controllable security parameter.

For the tape-space explosion to be effective, at least some set of the tape symbols must be used to initialize the tapes. Since the tape is not bounded in size, we must either use one symbol to denote a blank (not yet used) tape cell or reveal to the executer a set of cell values that we consider to be equivalent to "blank". Since there is no improvement in security by revealing multiple equivalent "blank" values, we instead let only one value denote blank for unused cells (we may have multiple "blank" equivalents in cells the Turing Machine populates or is populated as part of input). If we only introduced symbols from the first, smaller alphabet when writing values on the tapes, then deduplication of transitions would be trivial. Thus, we ensure that the write operations of the tape value σ are transformed into one of $\sigma', \sigma'', \sigma''', \cdots$.

5.3.3 Non-uniform blow-up

The explosion factor for any state or tape value should be determined by the distribution of that state or tape value during runtime. For instance, if one state was rarely attained – say it could only be attained via an unlikely execution path over the space of inputs – then having many shadow states for it would only minorly change the distribution of states achieved for the input space. If some state was highly symptomatic of a likely partition of inputs, then having many shadow states for this original state would create a more uniform distribution of achievable states during runtime. A similar logic can be applied to tape values. For instance, imagine a Turing Machine that reads some value α

Algorithm	Input bits	Alphabet Bitsize	States	Transitions	Oblivious
AND	512	2	2	5	True
COMP	512	2	4	19	True
MULT	128	2	10	72	True
HAM	256	2	6	19	True
PAL	512	6	3	8257	False
KNAP	50	3	12	23553	False

Figure 5: Algorithms implemented using the BlindTM system.

from the initial head position, then changed the value at α cells left of the starting head position to be α, writing back the same value read for all other encountered tape cells. To make this Turing Machine oblivious, discussed in §5.2, let the Turing Machine always move, in total, to a fixed number β of cells to the left such that $\forall_\alpha : \beta > \alpha$. When this Turing Machine is run, without tape-space explosion, the value of the first cell is trivial to determine. Tape space explosion helps mitigate this issue by changing the values written during the steps that move the head to α left cells. Since the Turing Machine does not allow the read cell to be stored as a variable, the writing back of the same value will be done via transitions of the form $q, \sigma \rightarrow q', \sigma, \mathit{left}$. After the tape explosion, this transition will look like $q, \sigma \rightarrow q', \sigma', \mathit{left}$, which is the same form that the final state that writes back α (or, in this case, the α-equivalent value) to the tape.

6. SYSTEM EVALUATION

To evaluate the BlindTM system, we first microbenchmark the runtime of each individual transformation from an unencrypted Turing Machine: homomorphic encryption, oblivious head movement, state/tape blowup. Next, we benchmark our system by measuring end-to-end runtime and comparing it with existing garbled circuit implementations on a set of common algorithms. Finally, we conclude with a discussion of improvements necessary to make the implementation more practical.

We constructed several programs in our Turing Machine language. Some algorithms were chosen to enable comparison with programs distributed with existing garbled circuit implementations, while others were chosen to demonstrate the flexibility of our programming paradigm.

- Bitwise AND (AND)
- "Millionaires" [13]/Unsigned Comparator (COMP)
- Binary Multiplication (MULT)
- Hamming Distance (HAM)
- Palindrome Detection (PAL)
- Knapsack Problem (KNAP)

Figure 9 shows the range of input sizes and state space required for each of these Turing Machines. We also note which computations are oblivious by nature, as these machines will not require additional oblivious transformation (§5.2).

For evaluation, a 64-bit 2.3 GHz processor with 8GB RAM was used with default Java Virtual Machine parameters.

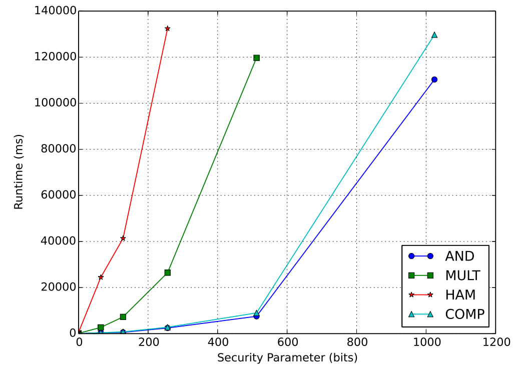
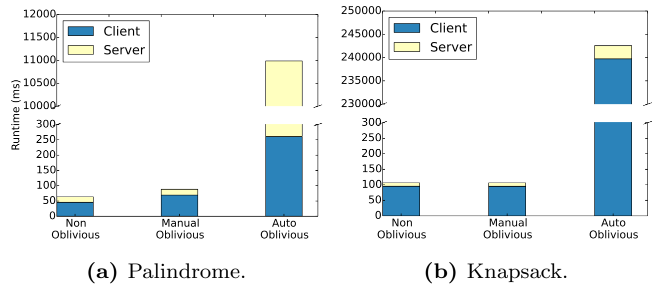


Figure 6: Runtime comparison of HPKEET with increasing security parameter. Largely dominated by modular exponentiation of client-side encryption and decryption.



(a) Palindrome. **(b)** Knapsack.

Figure 7: Comparison of runtime between two methods of oblivious head movement.

Server time denotes the time to evaluate the Turing machine, and client time denotes the time to construct the Turing Machine and retrieve the results from the returned results (as shown in Figure 2). All runtimes are averaged over three trials with a fixed seed to the pseudo-random number generator. Any results omitted indicate they did not complete within our timeout threshold of 10 minutes.

6.1 Homomorphic Encryption

Figure 6 shows the total (client and server) runtime of various algorithms as the security parameter increases from an unencrypted Turing Machine to 1024 bits. Recall that the security parameter is the number of bits of the prime number p that defines the group used for encryption. As the security parameter increases, security is enhanced, because the encryption becomes computationally harder to break. However, the client-side computations to encrypt and decrypt messages dominate the run-time due to the expensive modular exponentiation operations.

6.2 Oblivious Head Movement

Oblivious head movement requires that the movements of the heads depend only on the length of the input, not on its content. This restriction is placed to ensure the evaluator does learn any side-information about the contents of the tapes or the algorithm being performed. Many of our implemented algorithms, specifically the ones most similar to circuit algorithms, are already oblivious by nature. In or-

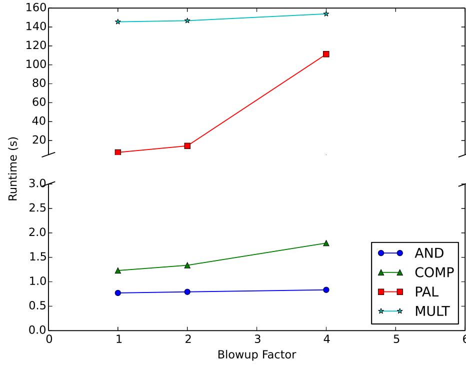


Figure 8: Runtime of auto-obliviated machines with increasing blowup factor.

der to better understand how the oblivious head movement requirement affects the runtime of the Turing Machine, we formed two machines for each problem: one in which the Turing Machine was manually made to be oblivious, and one in which the Turing Machine was automatically made oblivious using the construction described in §5.2. We performed this experiment for both our Palindrome and Knapsack programs.

As shown in Figure 7, our results indicate that manually constructing an oblivious algorithm results in almost no change in the runtime taken by the Turing Machine. However, the simple $O(n^2)$ algorithm adds a significant memory and runtime overhead.

We also note the sharp increase (three orders of magnitude) in the number of steps taken by the evaluator in running the Turing Machine, i.e. the number of transitions taken.

	Knapsack	Palindrome
Non oblivious	506	2050
Manual oblivious	546	3074
Auto oblivious	514602	8415250

Figure 9: Number of TM transitions taken by evaluator under different methods of oblivious transformation.

These results suggest that the oblivious head movement security guarantees are reasonable if a programmer is able to write their algorithm in an oblivious manner. This is often possible by decomposing the problem into smaller oblivious subtasks, and thus the programmer can avoid the overhead required by auto-oblivious transformation.

As mentioned in §5.2, there is a construction to generate an oblivious Turing Machine that runs in $O(n \log n)$ by Pippenger and Fischer, which is proven for a single tape Turing Machine. For future work, we hope to adapt the Pippenger and Fischer construction to multi-tape Turing Machines and measure its relative efficiency.

6.3 State/Tape Blowup

State/Tape blowup is also necessary to ensure that running the machine does not leak information about the contents of the tapes to the evaluator. Without blowup, an attacker

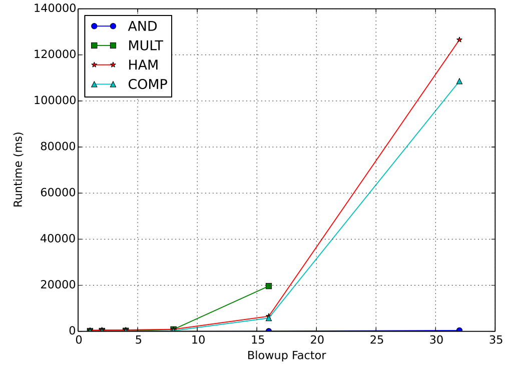


Figure 10: Comparison of runtime with increasing parameters of state/tape blowup. MULT timed out for blowup factors greater than 16.

could gain information about the distribution of time spent in states and/or the symbols contained and being written to the tapes.

Figure 10 demonstrates the effect of blowup as the parameter increases from 1 (no blowup) to 32, increasing exponentially. We find that even for Turing Machines with relatively large numbers of states and transitions, such as Multiplication, the blowup introduce minimal overhead up to a factor of 8. However, when combined with other transformations, blowup does exacerbate inefficiencies in runtime, as shown in Figure 8.

6.4 End-To-End Measurements

6.4.1 *Combining All Transformations*

Due to the large overhead incurred with auto-oblivious head movement, we perform end-to-end tests of our system with Turing Machines that are already constructed in an oblivious manner.

Figure 11 indicate the runtime with various increasing values of the security parameter and blowup factor. These results are encouraging because they demonstrate the feasibility of executing a Blind Turing Machine with all the necessary security guarantees and a runtime on the order of minutes. However, these results in isolation are difficult to understand, and so we compare our runtimes with the current state of the art method for secure function evaluation – garbled circuits – on a set of common algorithms.

6.4.2 *Garbled Circuits Comparison*

Figure 12 shows a comparison of BlindTM with two well-known garbled circuits implementations, Fairplay and FastGC. We compare with algorithms that were distributed as examples with these systems. We note that Fairplay uses a security parameter of 160 bits, whereas the newer FastGC system uses a security parameter of 1024 bits. These results indicate the BlindTM performs comparably to Fairplay, a first-generation garbled circuits implementation. However, BlindTM presents a higher overhead as compared with the newer garbled circuit implementation FastGC, even with BlindTM using a significantly smaller security parameter.

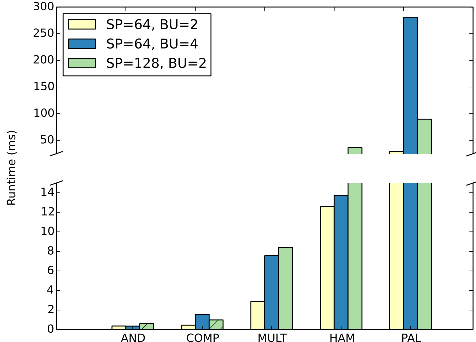


Figure 11: End-to-end runtime of programs. Includes homomorphic encryption, blowup, and oblivious head movement.

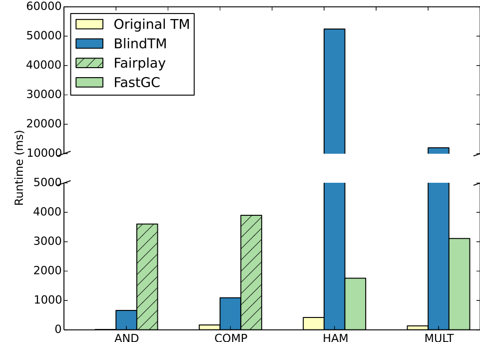


Figure 12: BlindTM comparison with garbled circuits. BlindTM trials used a fixed security parameter, blowup factor of 160 bits and 2, respectively.

This is primarily due to the poor scalability of HPKEET; mitigations of this factor are described in §6.5.

These results represent a preliminary comparison between the garbled circuits and BlindTM model in that the tested computations are easily expressible in circuits and thus more likely to perform well in the garbled circuits construction. We believe the Turing Machine programming paradigm is more flexible and enables easier expression of more complex algorithms such as sorting a list or dynamic programming algorithm such as knapsack. In future work, we plan to implement such algorithms in garbled circuits to allow for a comparison.

Finally, it should be noted that theoretical optimizations of garbled circuits have been published since the last release of FastGC or Fairplay, but the authors were unable to find an open-implementation of such systems.

6.5 Future Improvements

The practicality of the BlindTM system is limited by two factors: the computational cost of using HPKEET for comparisons and the storage/representation overhead of using a Turing Machine as described in §5. While automatically creating an Oblivious Turing Machine, incurs a high overhead, it can be mitigated by manually designing a Turing Machine with the oblivious property. In addition to these obstacles, the addition of a layer that transforms more common imperative languages into a Turing Machine may make the BlindTM system easier to adopt. While many of the costly parts of Turing Machine evaluation could be partially solved in hardware, the focus of the following is to describe improvements that use off-the-shelf computer processors.

The computational cost of HPKEET is extreme as can be seen in Figure 12. While HPKEET provides stronger security guarantees, the cost of these guarantees may outweigh the value they provide. One solution would be to modify the HPKEET method to make it more practical, possibly using a different encryption scheme, such as Paillier encryption [10] or elliptic curve cryptography [9].

Another method of mitigating the cost of HPKEET equality tests is by removing HPKEET entirely. By using a larger blowup factor for the state and tape blowup, the guarantees of HPKEET, excluding the need for the comparison token, can be provided. This mechanism has the advantage that the Turing Machine comparisons for state change require no computation and are just retrieval. Unfortunately, this increased blowup factor – needing the original inputs to be a negligible size with respect to the blownup space – exacerbates the overhead of the Turing Machine representation.

The representation of the Turing Machine as a direct mapping of state and input symbols to next state, output symbols, and head movements allows the state/tape blowup to be effective. Unfortunately, while the blowup factor does not change the asymptotic complexity of the Turing Machine, the practical runtime is effected by the increased memory usage as can be seen in Figure 10. Naively, one could construct a small symbolic function that takes the inputs and maps them to outputs. This mechanism would also allow for a more natural method of programming Turing Machines - instead of creating a transition table representing all possible combinations of “a” and “b” and mapping them to “a+b” for an addition operation, the programmer could just express “a+b” directly. Unfortunately, this method would prevent state and tape blowup from being effective. The reason that state and tape blowup is particularly effective is that the evaluator cannot determine a pattern between the inputs and outputs as they are randomly reassigned. If some abstract function were to be used to represent the control logic, then that logic would reveal information about the inputs. Indeed, the method of defeating the state/tape space blowup is by deduplicating values that are effectively the same. Any method that allows for compression of the control logic would be effectively deduplicating entries, or otherwise grouping entries by shared characteristics. Ideally, the control logic table would be random in appearance to the evaluator.

While a symbolic function representation may not ensure the security guarantees of this project, this does not mean that a higher level language could not be transformed into a Turing Machine. One simple mechanism to do this transformation would be to allow the programmer to write functions for

the control logic which are them evaluated on all possible inputs before being transformed by the BlindTM system. Another mechanism to transform higher level languages to the Turing Machine representation here could be attempted by using multiple tapes as registers or using multiple tapes as stacks in a stack machine. Unfortunately, there is one key programming characteristic would not be easy to represent in a Turing Machine: arbitrary, constant-time access via pointer. While modern architectures have a fixed number of registers, which a Turing Machine could also have, the ability to perform $O(1)$ lookups by pointer in an arbitrarily-sized memory would not be possible by the construction used here. An alternative construction would be necessary to account for the "head" movement/access pattern should a RAM system be used.

7. CONCLUSION

This paper has described BlindTM, the first implementation of a Blind Turing Machine. BlindTM is a system designed for secure function evaluation (SFE) which models computation as Turing Machine. A client constructs their algorithm and data as a Turing Machine and use a series of transformations (HPKEET, State/Tape Blowup, and Oblivious Head Movement) to encrypt the Turing Machine. This "blind" Turing machine can then be sent to an evaluator to be run, and returned to the client while still encrypted. We believe a secure and flexible system for SFE will be of increasing importance as cloud services/PAAS grow in popularity.

There are several different directions to explore from this work: the parameter space could be explored to find optimal space/security trade-offs, further benchmarks that leverage the way that Turing Machines handle memory could be devised, and automating detection of Oblivious Turing Machines could be explored for example. The speed of the Blind Turing Machine, while not faster than existing garbled circuit implementations, compares with the speed of the first-generation garbled circuits implementation tested, which makes the efficiency promising, though our system is not yet sufficient to supplant the position of garbled circuits as the standard means of SFE.

While exploring the security properties of the Blind Turing Machine model, we realized that the encryption provided by the HPKEET scheme does not protect against all vectors from the evaluator. The HPKEET phase primarily guarantees security against attackers that do not have the authorization token, which may limit applicability. Furthermore, the overhead of using the HPKEET encryption is large. Future directions of research include other implementation options, such as alternate HPKEET methods, or elimination of the HPKEET phase while maintaining security through the oblivious transformation and blowup phases. This may yield an implementation that is comparably efficient with modern garbled circuit implementations.

8. REFERENCES

[1] I. Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 445–456, London, UK, UK, 1992. Springer-Verlag.

[2] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[3] S. Goldwasser, Y. Kalai, R. Popa, V. Vaikuntanathan, and N. Zeldovich. How to run turing machines on encrypted data. In R. Canetti and J. Garay, editors, *Advances in Cryptology âĂŞ CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 536–553. Springer Berlin Heidelberg, 2013.

[4] W. Henecka, S. Kögl, A. reza Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: Tool for automating secure two-party computations. In *In ACM Conference on Computer and Communications Security (ACM CCS'10*, pages 451–462, 2010.

[5] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[6] H. Lipmaa. On the CCA1-security of Elgamal and Damgård's Elgamal. In *Proceedings of the 6th International Conference on Information Security and Cryptology*, Inscrypt'10, pages 18–35, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] L. Malka. Vmcrypt: Modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

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

[9] V. S. Miller. Use of elliptic curves in cryptography. In *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[10] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[11] N. Pippenger and M. J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, Apr. 1979.

[12] S. Rass. Blind turing-machines: Arbitrary private computations from group homomorphic encryption. *CoRR*, abs/1312.3146, 2013.

[13] A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[14] A. C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.