2011 Ninth Annual International Conference on Privacy, Security and Trust

General Secure Function Evaluation Using Standard Trusted Computing Hardware

Stephen R. Tate
Department of Computer Science
University of North Carolina at Greensboro
Greensboro, NC 27402
E-mail: srtate@uncg.edu

Roopa Vishwanathan
Dept. of Computer Science and Engineering
University of North Texas
Denton, TX 76203
E-mail: rv0029@unt.edu

***Abstract*—In this paper, we show how Trusted Platform Modules (TPMs), standard security hardware devices, can be used with minor modification to efficiently support Secure Function Evaluation (SFE), a fundamental and extremely powerful cryptographic operation. Prior research by others has shown how SFE can benefit from using security hardware, but prior work has used either custom hardware tokens or powerful secure co-processors which require significant changes to current computing systems. In this paper we show that similar techniques can be supported by TPMs with enhancements that are at the level of a firmware upgrade (albeit a secure firmware upgrade endorsed by the TPM manufacturer) — specifically, no new physical devices would need to be purchased or added to most modern business-class systems. This paper describes the specific changes that need to be made, and evaluates efficiency for a simple example in location-based privacy. Our evaluation shows that performance is reasonable for supporting simple privacy-enhanced applications.**

I. Introduction

Secure Function Evaluation, or SFE, is a fundamental cryptographic problem in which each of $n \geq 2$ parties has an input x_i (for $1 \leq i \leq n$) and the parties wish to compute some function $f(x_1, x_2, \cdots, x_n) \rightarrow (y_1, y_2, \cdots, y_n)$ such that party i learns output y_i, and no party gains any information about the other inputs or outputs other than what follows from the output that it receives. There are no trusted parties, either among the participants or as a trusted third party, since a trusted party could simply receive the inputs, compute the function, and distribute the outputs to the appropriate parties. The SFE problem can be solved for any polynomial-time function f by first representing f as a polynomial-size boolean circuit, and then converting this into a "garbled circuit" that can be obliviously evaluated using garbled (i.e., encrypted) wire values — this idea was first introduced by Yao [1], and general-purpose solutions for the SFE problem typically follow this technique (more recently, homomorphic encryption schemes offer a different approach, but with a different trust model [2]).

The two most common adversary models in cryptography are the semi-honest model (sometimes called "honest-but-curious," in which the parties follow the prescribed protocol, but try to learn information which they are not entitled to) and the malicious model (in which the parties may deviate from the prescribed protocol). Garbled circuits were first introduced by Yao [1] as a solution for SFE in the two-party semi-honest model, and the notion was extended by Goldreich, Micali and Wigderson (GMW) [3] to the case of $n > 2$ semi-honest parties. If there is even a single semi-honest party, this party can construct the garbled circuit, each party can get the garbled version of their input using oblivious transfer [4], and after distributing the circuit and garbled inputs, each party can evaluate the circuit to obtain their output. In several important and practical applications of SFE, including using garbled circuits for auctions [5] or mobile agents [6], [7], the trust model of the application allows us to assume semi-honest behavior by the circuit constructor, and so these basic garbled circuit construction techniques are sufficient.

If there is no semi-honest party to construct the circuit, then SFE becomes significantly harder. Consider, for example, a two-party protocol between Alice and Bob that determines a "winner" (e.g., two parties seeing who holds a higher value, while the values remain secret). If Alice constructs the circuit and is dishonest, she can create a garbled circuit which looks genuine, but in which the final output gate always outputs that Alice is the winner. Any multi-party application in which the parties have no reason to trust each other, and the parties play equivalent roles (so there is no natural "leader") has a similar problem. Beaver, Micali and Rogaway solved the SFE problem for multiple parties in which up to $n/2$ parties may be malicious [8], using a collaborative circuit construction with communication complexity $\Theta(n^2 G)$, where G is the number of gates in the circuit. There have been multiple variations proposed for the malicious adversarial model of SFE; some representative work includes [9], [10]. Almost all of the solutions for the multi-party malicious model are based on either Goldreich's compiler [3], or are based on zero knowledge proofs attesting to the honest construction of the garbled circuit. The compiler of Goldreich uses generic zero knowledge proofs (ZKPs) which are mainly of theoretical importance and are quite inefficient to apply in practice. Cut-and-choose is a popular paradigm used in interactive ZKPs and while it isn't as expensive as generic zero knowledge proofs, is still a highly interactive construct and significantly increases the communication costs of the protocol using it.

In the past few years, there has been increased attention given to the possibility of adding new security hardware to

978-1-4577-0584-7/11/$26.00©2011 IEEE

standard computing platforms, both in basic research and in practice. Looking specifically at SFE in the presence of malicious adversaries, Järvinen *et al.* propose a solution using a custom-built tamper-proof token, which results in trustworthy circuit construction and also significantly reduced communication complexity [11]. Sadeghi *et al.* use similar ideas in the context of outsourcing computation to cloud computing resources — this new application model has different trust characteristics from Järvinen, and leads to additional insights [12]. In both of these cases, systems must be augmented with hardware that is not present in systems today — Järvinen *et al.* describe a custom-built hardware token (which they evaluate using an FPGA implementation), and Sadeghi *et al.* consider both a custom token or a heavy-weight secure coprocessor such as the IBM 4758/4764.

In this paper, we consider using Trusted Platform Modules (TPMs) to enhance SFE. TPMs are security hardware devices that have become standard in business-class systems (laptops and desktops), and so are already widely deployed. A variety of manufacturers make TPMs, but all conform to standards defined by the Trusted Computing Group [13], and experiments show that performance is similar across different manufacturers, and can accurately be estimated for new functionalities and protocols [14]. While standard TPMs do not support SFE, we can ask the following questions: What enhancements would be required for TPMs to support SFE? What efficiency could be expected in such operations? Exploring these questions in this paper, we show that enhancements are minimal — unlike previous SFE work using trusted hardware, no new hardware is required for most systems, and the changes are comparable to a firmware upgrade to hardware that is already widely deployed (a more detailed comparison to previous work is given in Section III). To evaluate efficiency, we consider a simple application in a location privacy setting, computing a group nearest neighbor, and find that the resulting efficiency is reasonable for applications such as this which are simple but useful. While TPMs are not powerful computing engines, we benefit from the fact that TPMs are only used in a precomputation phase for circuit construction, and only use very efficient cryptographic operations (which are fast, even on computationally weak TPMs), while the time-sensitive circuit evaluation uses standard non-TPM garbled circuit evaluation running at full speed on the CPU.

As in previous research that uses secure hardware to support SFE, our main benefit is that the SFE protocol is protected from malicious adversaries providing dishonest garbled circuits, without relying on expensive cut-and-choose zero knowledge proofs. In essence, we replace the statistical trust gained through cut-and-choose with a trust based in trusted hardware. To gain this benefit, a modified TPM is used in the circuit construction phase. Garbled circuit creation can take place well in advance of the use of the circuit, and is in general not a time-sensitive operation (while circuit evaluation might be, and we use the standard non-TPM-based evaluation algorithms). To support garbled circuit creation, we only need to add four new commands to a TPM. The first three (TPM_SFEINIT, TPM_SFEINPUTS, and TPM_SFEGATE) use only symmetric cryptography (in particular, hashing), and are very fast even on devices as limited as TPMs. The last operation, TPM_SFEFINISH, requires a digital signature, which can take around half a second on current TPMs, but this is only performed once per circuit construction. These new TPM operations use only cryptographic functionality that is already present in TPMs for other purposes, so adding these commands is simply a matter of connecting the right pieces and providing an appropriate interface — if these changes were adopted as standard TPM functionality, then the changes are essentially a TPM firmware update, and don't require any new hardware.

II. Trusted Platform Modules

In this paper, we focus on the use of Trusted Platform Modules (TPMs) in supporting SFE. TPMs are hardware components that are based on specifications developed by the Trusted Computing Group, an industry consortium of over 100 companies [13]. TPMs provide a variety of functions to support system and application security, including the ability to keep and securely report measurements of system parameters (BIOS, hardware elements, operating system, software applications, among others) to provide a measured and protected execution environment, which assures the integrity and trustworthiness of the platform. In addition measurement capabilities, TPMs support a variety of standard cryptographic capabilities, including random number generation, public key functionality for encryption and digital signatures (using RSA), cryptographic hashing (using SHA1), and message authentication codes (using HMAC-SHA1), as well as more specialized operations such as a privacy-oriented authentication technique known as direct anonymous attestation. To support these operations, TPMs include internal *protected storage* locations, used for storing private keys and other sensitive information needed to perform TPM functions.

Summarizing the capabilities and security features of an ideal TPM into high-level observations, we provide the following formalization of TPM security; we point out that this definition reflects the real-world design goals as expressed in the standard TPM Protection Profile [15].

Definition 2.1 (Trusted Platform Security Assumption): The *Trusted Platform Security Assumption* is the assumption that the system containing a TPM satisfies the following properties:

1) *Tamper-resistant hardware:* It is infeasible to extract secrets stored in protected locations in the TPM.
2) *Secure Encryption:* The public-key encryption algorithm used by the TPM is CCA-secure.
3) *Secure Signatures:* The digital signature algorithm used by the TPM is existentially unforgeable under adaptive chosen message attacks.
4) *Trustworthy PrivacyCA:* Only valid TPM-bound keys are certified by a trusted PrivacyCA.

III. Related Work

Hardware-assisted security in the design of cryptographic protocols has garnered much attention and investigation recently. There has been a fair amount of recent research in realizing various cryptographic functionalities using the idea of trusted computing, whether abstract hardware tokens or more concrete instantiations such as TPM chips. The idea of hardware-assisted security can be traced back to the e-cash scheme due to Chaum and Pederson where hardware tokens facilitate secure transactions between customers and a bank [16], and Goldreich and Ostrovsky's oblivious memory model (or ORAM) which enables software to run in the presence of untrusted memory [17]. This idea was revisited recently by Katz, who studied the problem of achieving universally composable security for certain two-party functionalities by using tamper-proof hardware tokens that are produced by each party [18]. Other recent work in the area of using abstract tokens for realizing cryptographic functionalities include [19], [20], [21], [22]. In addition to work using general secure hardware, several previous results have considered specifically the use of TPMs in cryptographic protocols, including for efficient non-interactive oblivious transfer [23] and for verifiable encryption [24].

In studying general SFE, Fort *et al.* consider how to use a smartcard to support SFE, and have described a peer-to-peer distributed system for secure multi-party computation where each peer maintains a secure module (or a smartcard) [25]. In contrast, we require that only the creator of the garbled circuit be equipped with a TPM. In other work, Iliev and Smith [26] gave a two-party SFE protocol using the IBM 4758 secure-coprocessor, a product which has since been discontinued, although the replacement (called the IBM PCIe 4764 crypto co-processor) could certainly be used, at a cost of as much as $10,000. In more recent work, Iliev and Smith [27] extend this work by presenting a compiler, which they name *FairiePlay*, for creating circuits. Sadeghi *et al.* [12] propose a scheme that can compute queries efficiently in a distributed environment, but requires a powerful secure co-processor. They do not give any implementation or real-world instantiation of the secure co-processor, but point out that the IBM 4764 series can be used. The expense of this co-processor (also noted by Iliev and Smith [27]) makes it unrealistic to expect regular users to purchase it.

In work that is closest to the results in this paper, Järvinen *et al.* consider the SFE problem in a two-party setting where a server issues a low-cost tamper-proof hardware token to a client [11]. While our work and that of Järvinen *et al.*'s progressed independently, both produced several of the same innovations. The most important of these is a garbled circuit construction that uses pseudo-random functions to allow construction with severely space-limited devices (previous garbled circuit construction techniques required secure space that is proportional to the width of the circuit, which can be quite large). One distinction is that Järvinen *et al.* create the hardware token on the server side and then and send it over to the client, whereas in our setting, the trusted computation (i.e., the use of the TPM) stays at the server's side. Our technique has the benefit of keeping all trusted hardware on one side, but as a result requires higher communication complexity (although Järvinen *et al.*'s cost of physical transmission of a piece of hardware should not be ignored!). We point out that if the client also has a TPM, then we can achieve the same reduced communication complexity by using the client's TPM to construct the circuit (with some additional safeguards).

IV. Hardware-assisted Secure Function Evaluation (HASFE)

In this section, we present a TPM-assisted protocol for SFE using garbled circuits, HASFE_GC, where we consider having two active parties: Alice, who acts as a circuit creator, and Bob, who acts as a circuit evaluator. One can potentially extend this to multiple parties, but with the condition that a party cannot take on the roles of a circuit creator and evaluator both. This can be enforced by checking the signatures on encrypted circuits: if a party's TPM signs a circuit, that party will not be allowed to be an evaluator (the group of parties can also maintain a list of potential evaluators or a "blacklist" of people who aren't eligible to be evaluators).

A. Hardware-Assisted SFE using Garbled Circuits

We consider a scenario where we have n participants who want to compute a function $f(x_1, x_2, \cdots, x_n)$ over their individual inputs $x_1, x_2, \cdots, x_n$ without any party revealing its input to other parties and in the absence of a trusted third party; for simplifying the discussion, we set $n = 2$. Since we consider the malicious adversarial model, we cannot trust Alice to create the circuit correctly. In solutions that have been proposed for the malicious model up until now, Alice would have had to provide a zero-knowledge proof at every stage of the circuit construction to other parties proving that she has constructed the circuit correctly. This is the part we replace with trust in secure hardware, or more specifically TPM chips. We use a slightly modified version of Kolesnikov and Schneider's circuit construction technique [28] for our circuit construction.

Let there be $n = 2$ parties each having m-bit inputs; let the output of the circuit be m-bits. Let the plaintext truth table that defines a gate g_i implementing functionality $\otimes$, with input wire indices, a, b and output wire index c be denoted by TT_{g_i}, which consists of $2^{\text{Num. of input wires}}$ entries — in this case 4 entries. $TT_{g_i} = \langle (0,0,0 \otimes 0), (1,0,1 \otimes 0), (0,1,0 \otimes 1), (1,1,1 \otimes 1) \rangle$. Let the circuit have W wires with $0, \cdots, nm-1$ being input wires and $W-m, \cdots, W-1$ being output wires. Let N denote the number of bits produced by the hash and HMAC functions in the TPM (e.g., 160 bits using SHA1), and so garbled signals for wires are N bits, as are garbled truth table entries. In constructing the garbled circuit, the TPM outputs each gate's garbled truth table by using a special TPM command. For doing this, we need a few simple extensions to the existing TPM specification to support the garbled circuit creation, something along the lines of an

"SFE suite" of commands. The following are the proposed TPM extensions for performing SFE using garbled circuits:

`TPM_SFEInit`$(gateNum, inputNum) \rightarrow (cktHandle)$:
Alice's TPM takes in the number of gates and input wires in the unencrypted boolean circuit, and initializes a structure in its protected storage in order to construct the circuit. The TPM then generates a secret key which it calls SFESeed, which will only be known to the TPM and will be used as a seed to a Pseudo Random Function (PRF) implemented using HMAC (it is known that HMAC is a good candidate for instantiating a PRF under the assumption that its compression function is a PRF [29]). Additionally, the TPM creates a random string, $R \in \{0,1\}^N$ which will be used for computing garbled wire signals. R needs to be generated in a way such that its least significant bit (lsb) is 1, for reasons that will be explained in the TPM_SFEGATE command. Finally, the TPM initializes two contexts for computing digests, one for the plaintext truth tables, $hctxPlain$, and one for the encrypted truth tables, $hctxEncr$. All of the above created variables are stored in an internal TPM structure called TPM_SFE_DATA which the TPM references in future SFE commands for this particular circuit:

```
TPM_SFE_DATA {
   HashContext hctxPlain,hctxEncrypted
   int numGates, numInputs
   TPM_DIGEST SFESeed,R
}
```

As the output of this command, the TPM returns the handle that serves as a reference to the internally stored TPM_SFE_DATA structure.

`TPM_SFEInputs`$(a, cktHandle) \rightarrow (k_a^0, k_a^1)$:
In this command, the TPM first checks whether a designates a valid input wire (i.e., whether $a < cktHandle.numInputs$), and if so it generates and outputs the corresponding input wire signals: $k_a^0 = HMAC(SFESeed, a)$ and $k_a^1 = k_a^0 \oplus R$. Note that Alice can infer the value of R from these values, and if R is known to the circuit evaluator there is no secrecy in the evaluation. It is for this reason that we require that one party (e.g., Alice) cannot be both a circuit creator and a circuit evaluator. This command needs to be called nm times by Alice in order to get the garbled input wire values for the entire circuit. At this point, there are two ways for Alice to handle the input values and send them to Bob:

1) Option 1: Alice performs a 1-of-2 oblivious transfer (or OT_2^1) [4] for each of Bob's m input bits. Note that if we have $n \geq 2$ parties, where the remaining parties other than Alice and Bob are passive participants, then they each have to get their garbled input bits from Alice via OT_2^1 and send their garbled inputs to Bob for evaluation. Bob, after receiving inputs from all parties evaluates the circuit.
2) Option 2: This option can be chosen if we have multiple circuit evaluators (anyone can be an evaluator except Alice), or if circuit evaluation needs to be done non-interactively so that standard oblivious transfer is not an option. Using a TPM extension proposed by Sarmenta *et al.* [30], Bob can create m one-time use keys as *count-limited objects* (*clobs*), certify the clobs, and send the certified clobs to Alice. Alice then uses each clob to separately encrypt the two input wire signals for each of Bob's input bits, and sends the ciphertexts to Bob. The count-limit on each key ensures that Bob can only decrypt one of the input signals, preserving this important property from the standard oblivious transfer option. We note that the non-interactive oblivious transfer technique of Gunupudi and Tate can achieve this same effect using only a single clob [23], and hence is much more efficient.

`TPM_SFEGate`$(a, b, c, cktHandle, g_i, TT_{g_i}, flag)$
$\rightarrow (ET_{g_i}, (e_0, e_1))$:
This command produces the encrypted truth table for gate g_i with truth table TT_{g_i}, with input wires a and b and output wire c, using the following steps.

1) This command checks that g_i is a valid gate number (i.e., that $g_i < cktHandle.numGates$), and if so then it computes the necessary wire signals: $k_a^0 = HMAC(SFESeed, a)$, $k_a^1 = k_a^0 \oplus R$, $k_b^0 = HMAC(SFESeed, b)$, and $k_b^1 = k_b^0 \oplus R$. Recall that the lsb of R is 1, so values k_i^0 and k_i^1 for any wire i always differ in their lsb. This lsb is a selector bit, used to select a row of a truth table, and for wire i we use p_i to denote the lsb of signal k_i^0. The same calculation is performed to get the wire signals for the output wire: $k_c^0 = HMAC(SFESeed, c)$ and $k_c^1 = k_c^0 \oplus R$.
2) Next, the TPM needs to create truth tables for the gate using the wire signals computed in the previous step. Specifically, the TPM computes the following four values, and places them in the specified positions:

   Position (p_a, p_b): $e_{0,0} = H(k_a^0 || k_b^0 || i) \oplus k_c^{0 \otimes 0}$
   Position $(\overline{p_a}, p_b)$: $e_{1,0} = H(k_a^1 || k_b^0 || i) \oplus k_c^{1 \otimes 0}$
   Position $(p_a, \overline{p_b})$: $e_{0,1} = H(k_a^0 || k_b^1 || i) \oplus k_c^{0 \otimes 1}$
   Position $(\overline{p_a}, \overline{p_b})$: $e_{1,1} = H(k_a^1 || k_b^1 || i) \oplus k_c^{1 \otimes 1}$

   where H is a hash function that is ideally modeled as a random oracle. The four garbled truth table entries are referred to collectively as ET_{g_i}, the garbled truth table for gate g_i.
3) The TPM checks if $(flag == true)$. The $flag$ is used for keeping track of the output wires. If

$(flag == true)$, it means that this gate is an output gate and c being the output wire of this gate, the TPM needs to output the values of the output wire encodings. The TPM computes:
$e_0 = H(k_c^0 || \text{"}out\text{"} || g_i) \oplus 0$ in position p_c
$e_1 = H((k_c^0 \oplus R) || \text{"}out\text{"} || g_i) \oplus 1$ in position $\overline{p_c}$
where p_c is the lsb of k_c^0. In case $(flag == false)$, the TPM just returns $\perp$ in place of the e_0, e_1 values.

4) Lastly, the TPM updates the plain and encrypted truth table digests in TPM_SFE_DATA by extending the hash contexts $hctxPlain$ with $g_i || TT_{g_i}$ and $hctxEncr$ with $g_i || ET_{g_i}$.

Finally, the TPM returns the values ET_{g_i} and (e_0, e_1).

`TPM_SFEFinish`$(cktHandle, aikHandle) \rightarrow (Sign_{AIK}(hashPlain, hashEncr))$:
This command is called after the TPM_SFEGate command has been executed for each gate. The TPM first closes out the hash contexts $hctxPlain$ and $hctxEncr$ to get the final hash values for the list of plaintext gates and garbled gates, $hashPlain$ and $hashEncr$, respectively. Then the TPM uses an AIK to sign the hash values as an attestation that it created the circuits that resulted in the given hash values.

The algorithms for the circuit creator and evaluator consist of calls to the above TPM functions in the obvious way. Due to space constraints we do not give them here.

Protocol HASFE_GC: Let Alice's inputs be $x = (x_1, \cdots, x_m) \in \{0,1\}^m$ and Bob's inputs be $y = (y_1, \cdots, y_m) \in \{0,1\}^m$. Let their common inputs be a plaintext circuit, $C(x,y) = f(x,y)$ where $f : \{0,1\}^m \times \{0,1\}^m \rightarrow \{0,1\}^m$ is the function that they wish to evaluate. The following is a hardware-assisted SFE protocol for securely computing f.

1) Alice constructs garbled circuit $C'(x,y) = f(x,y)$ using **Algorithm** GCCreator.
2) Alice sends her garbled input wires: $(k_{x_1}, \cdots, k_{x_m})$ to Bob and also sends the encrypted truth tables to Bob.
3) Alice sends Bob his garbled inputs in one of two ways:
   a) Alice and Bob engage in OT_2^1 so that Bob receives $(k_{y_1} \cdots, k_{y_m})$
   b) Bob either generates m one-time use *clobs*, or generates a single m-time use *clob* of the form: $\alpha = (PK, SK, countLimit)$, as the case might be, and sends them to Alice who sends Bob his encrypted inputs.
4) Bob runs **Algorithm** GCEvaluator and outputs $f(x,y)$, the output wires of the circuit.

We first prove that the HASFE_GC protocol is secure in the presence of a semi-honest Alice and semi-honest Bob and then prove that if Alice and Bob are malicious, the proof still holds. We assume, according to Bellare's paper [29] that HMAC is a PRF if its compression function is a PRF.

Theorem 4.1: If we use a TPM that satisfies the Trusted Platform Security Assumption, and assuming that HMAC can be used to instantiate a PRF, the HASFE_GC protocol is secure in the presence of malicious PPT adversaries.

Proof: There are two parts to the proof: One is the case where the circuit creator is malicious and the other is the case where the circuit evaluator is malicious. We assume that we use a secure OT_2^1 protocol and there exists a OT_2^1 simulator for Alice and Bob. We first prove the protocol secure in the semi-honest model and then show how to extend the proof to the malicious model.

Case 1 Alice is corrupted: Let there be a simulator A that simulates Alice's view of the protocol. A needs to produce Alice's output given the input $A(x, f(x,y))$. A simulates the garbled circuit in the same way as Alice and outputs the garbled input values and encrypted truth tables. A then runs the OT_2^1 simulator with fake garbled inputs corresponding to Bob and outputs the transcript of the OT_2^1. It is easy to see that the output of Alice and A are computationally indistinguishable from each other.

Case 2 – Bob is corrupted: In Kolesnikov and Schneider's paper [28], the garbled circuit evaluation was proven secure with respect to a semi-honest Bob using *truly random strings*. In this proof, we need to do the same using a pseudo-random function (PRF) generated using a HMAC instead of truly random strings. So, we need to prove that if a corrupted Bob cannot gain any advantage against an Alice who uses truly random strings, he cannot gain any advantage against an Alice who uses PRF-strings generated by her TPM either.
Let there be two simulators for Bob: B and B' such that B works with HMAC-PRFs and B' is given access to a true random oracle. We show that one cannot construct a polynomial-time distinguisher D that can consistently distinguish with non-negligible probability between the outputs of B and B'. We describe B and B' below:
Simulator B: Let B be a simulator that simulates Bob's view of the protocol, B needs to produce Bob's output given the input $B(y, f(x,y), \mathcal{HPRF})$ where $\mathcal{HPRF}$ is a HMAC-PRF that generates pseudo-random strings. B has access to Bob's input, but the real Bob gets the garbled circuit from Alice as well, which B does not receive as input. So, B needs to simulate the entire garbled circuit with garbled inputs and garbled truth tables as received by Bob. For simulating the circuit, B generates a random string $R' \in \{0,1\}^N$ and generates the garbled inputs thus: $(k_1^0 \leftarrow \mathcal{HPRF}(1), k_1^1 \leftarrow k_1^0 \oplus R', \cdots, k_{nm}^0 \leftarrow \mathcal{HPRF}(nm), k_{nm}^1 \leftarrow k_{nm}^0 \oplus R')$. It also generates fake truth tables thus: $e'_{00} = H(k_a^0 \parallel k_b^0 \parallel i) \oplus k_c^{0 \otimes 0}$, and so on for the other three table entries.

The table entries are easily placed in position based on the lsb's of the input wires. B then creates fake output tables using the same procedure as in Kolesnikov and Schneider's proof [28], i.e., for each circuit output wire, W_i, create a fake garbled output table with both entries corresponding to the same output garbled value:

$$e'_0 = H(k_c'^0 || \text{"}out\text{"} || g_i) \oplus f_i(x,y)$$

$e'_1 = H((k'^0_c \oplus R')||\text{"}out\text{"}||g_i) \oplus f_i(x,y)$

Finally B outputs $Sign_{AIK}(hashPlain, hashEncr)$ - the digests of the plaintext and encrypted circuits. Here we require that B should be able to simulate both, the TPM's AIK and the PrivacyCA certifying the AIK, since Definition 2.1 stipulates that TPM signatures are existentially unforgeable.
For simulating the transfer of garbled input values, B either runs the OT^1_2 simulator with its inputs $(y_1, \cdots, y_m)$ and outputs the transcript of the OT^1_2, or uses a *clob* as the case might be. If B needs to use a *clob*, and simulate the TPM's signature on it, B needs to simulate both, the TPM's AIK and the PrivacyCA that certifies the AIK. In the end, B returns the signed hashes, the fake tables, inputs, and fake output tables. Hence B, with access to a HMAC-PRF, can correctly simulate the view of Bob such that the output of B is computationally indistinguishable from that of Bob.
Simulator B': Let B' be a simulator that simulates Bob's view of the protocol, B' needs to produce Bob's output given the input $B'(y, f(x,y), \mathcal{RO})$ where $\mathcal{RO}$ is a random oracle that generates true random strings. B' needs to simulate the entire garbled circuit as received by Bob. For simulating the circuit, B' generates a random string R'' and generates the garbled inputs thus: $(k^0_1 \leftarrow \mathcal{RO}(1), k^1_1 \leftarrow k^0_1 \oplus R'', \cdots, k^0_{nm} \leftarrow \mathcal{RO}(nm), k^1_{nm} \leftarrow k^0_{nm} \oplus R'')$ and the garbled tables too, just like B does. For simulating the transfer of garbled input values, B' either runs the OT^1_2 simulator with the inputs it is given $(y_1, \cdots, y_m)$ and outputs the transcript of the OT^1_2, or if it needs to use a *clob*, B' generates the *clob(s)* and encrypts the garbled input values in the same way as B did.
Hence B', with access to a random oracle, can correctly simulate the view of Bob such that the output of B' is computationally indistinguishable from that of Bob.
Assume that we have a polynomial-time distinguisher D who can distinguish between the output of B and the output of B'. The following is the standard PRF-game played by D:
Game PRF

1) D is given access to a PRF-oracle: $\mathcal{PO} \in \{\mathcal{PRF}, \mathcal{RO}\}$ where the $\mathcal{PRF}$ is generated from a HMAC. D asks $\mathcal{PO}$ to output a set of garbled inputs.
2) If $\mathcal{PO} = \mathcal{PRF}$, it constructs the garbled inputs in a way similar to B, if $\mathcal{PO} = \mathcal{RO}$, it constructs the garbled inputs in a way similar to B'.
3) D examines the garbled inputs and outputs a guess g

D wins the game if it can distinguish between the case when $\mathcal{PO} = \mathcal{PRF}$ and the case when $\mathcal{PO} = \mathcal{RO}$. The advantage of D winning the PRF game can be bounded as:

$$Adv_{PRF}(D) \leq Adv_{HMAC}(D)$$

Since we assume, according to Bellare's paper [29] that a HMAC is a PRF it its compression function is a PRF, for every positive polynomial $p(\cdot)$ and all sufficiently large n's, the advantage of D winning the PRF game can be bounded as:

$$Adv_{PRF}(D) = Pr_{g \leftarrow \mathcal{PRF}}[D^g = 1] - Pr_{g \leftarrow \mathcal{RO}}[D^g = 1] < \frac{1}{p(n)}$$

where the probabilities are over the choices of g and the coin tosses of D. If such a D exists, it can distinguish between B and B' in the circuit construction process, since B uses $\mathcal{PRF}$ to create the garbled circuit and B' uses $\mathcal{RO}$ to create the garbled circuit. The advantage of D when given a random member of the set $\{B, B'\}$ is:

$$Adv_D(B, B') \leq Adv_{PRF}(D)$$

Since the $Adv_{PRF}(D)$ is negligible in the PRF-game, $Adv_D(B, B')$ is also negligible. Hence B and B' are computationally indistinguishable from each other. From Kolesnikov and Schneider's construction [28], B' which uses true random strings in the circuit construction process is secure in the semi-honest model. Since B and B' are computationally indistinguishable, B, which uses a HMAC-PRF in place of a true random oracle is also secure in the semi-honest model.

Extending the result to the malicious model: In the malicious model, the main adversary to protect against is a malicious Alice. Traditionally, Alice does a cut-and-choose zero knowledge proof to prove that the circuit inputs were suitably randomized and they were used in generation of the truth tables, etc. Here, Alice's circuit is completely constructed by her TPM. We had shown in Case 1 that a semi-honest but corrupted Alice can be protected against (there exists a simulator A, which on inputs $A(x, f(x,y))$ can produce Alice's view of the protocol). The only thing that a malicious Alice can do that a semi-honest Alice cannot do is to present a circuit to Bob that *appears* to have been certified by her TPM, but in reality the TPM does not certify it. The only way for a malicious Alice to do this is by forging the TPM's AIK signature over her encrypted circuit:

$$Adv_{HASFE}(Alice) \leq Adv_{AIK}(Alice)$$

From Definition 2.1, the signature algorithm used by the TPM is existentially unforgeable, so $Adv_{AIK}(Alice)$ is negligible and hence $Adv_{HASFE}(Alice)$ is also negligible.
Bob can be malicious, but since we are considering the model where there is a single circuit creator and no circuit creator can also be an evaluator, a malicious Bob cannot do anything that he could not have done in the semi-honest model. Hence the proof. ∎

V. Performance Evaluation

We have developed the HASFE_GC protocol using TPMs, but in principle, it can be implemented on any form of trusted hardware such as smartcards or secure co-processors, as long as we have a trusted execution environment. For implementing the HASFE_GC protocol on TPMs, we require TPM extensions equivalent to a firmware upgrade. In order to explore other forms of trusted execution environments which may not necessarily need extensions, we have implemented and compared the circuit creation time of the HASFE_GC protocol first on a simulated TPM, and next on a trusted computing infrastructure known as *Flicker* [31].

Flicker is an execution infrastructure developed by McCune *et al.* that leverages the capabilities of a TPM chip such as sealed storage and attestation with the *late launch* capability provided by either AMD's new SVM (Secure Virtual Machine) technology or Intel's TXT (Trusted Execution Technology). The advantage of using Flicker is that the trusted environment code-base of Flicker is just 250 lines, and we do not need extensions to current TPMs for running security-sensitive code on Flicker. The disadvantage of Flicker is that the OS is suspended during the execution of Flicker and the application needs to be small enough and needs to execute quickly in order for the OS to resume. Another disadvantage is that the overhead for setting up and shutting down a Flicker session is still quite high ($\approx$3-4 seconds), hence it cannot be used in applications where speed is a priority. Subsequent work by some of the creators of Flicker demonstrated a new system called TrustVisor which achieves the same goals but with a dramatic improvement in performance [32]. TrustVisor is too recent to have been tested in the current work, but this will be explored in future work.

For our experiments using a simulated TPM, we used TPM/J, a Java-based front-end interface developed by Sarmenta et al. [33], for communicating with the TPM simulator. The simulator is based on version 0.6 of Mario Strasser's TPM simulator [34], using timing-accurate extensions due to Gunupudi and Tate [14]. The experiments required that we add the four new SFE commands, TPM_SFEINIT, TPM_SFEINPUTS, TPM_SFEGATE and TPM_SFEFINISH. The timing-accurate extensions of the TPM simulator allow the use of timing profiles based on real TPMs manufactured by Atmel, ST-Microelectronics, Infineon, and Winbond, and for the experiments described here we used the Infineon timing profile since that represents a solid wide-spread TPM. While we could have used other timing profiles as well, our main goal was to compare the performance of the TPM and Flicker for the HASFE_GC protocol, and not to compare the performance of different TPMs with each other.

A. Application to location-based services

For evaluating the performance of the HASFE_GC protocol, we needed to pick an application where it could be potentially used. Location-based services are available on a wide variety of mobile platforms such as cell-phones, PDA's, and GPS systems, and popular social networks such as Facebook, Google Buzz, Foursquare, etc. have recently begun offering location services to their subscribers. As an example application of the HASFE_GC protocol, we consider the situation where a group of n parties want to jointly compute a common location of interest, their group nearest neighbor, without revealing their individual locations to each other and in the absence of a trusted third party. We transform the group nearest neighbor function into its boolean circuit representation and pick a user Alice who acts as the garbled circuit creator, and a user Bob who acts as the evaluator. Among the parties, only Alice needs to be equipped with a TPM chip and/or Flicker. Any party can be corrupted by an adversary, including Alice and Bob, and none of the parties trust each other.

We chose a small group of 10 parties (note that the number of parties doesn't really affect the circuit computation and evaluation time) and varied the number of locations from 500–2000. We used 64-bit values for (x,y) coordinates, specifically (latitude, longitude) values obtained from a real-world California road network dataset provided by Li *et al.* [35]. The circuit consisted of 500–2000 ripple-carry adder gates, each consisting of 64 1-bit full-adder gates connected in series, in addition to a single 64-bit comparator gate. The total number of gates in terms of composite gates varies from 501-2001 gates for 500–2000 locations.

B. Individual Command Times

We first report the time taken for executing individual commands on the TPM simulator and Flicker. The TPM_SFEINIT and TPM_SFEFINISH commands are one-time costs while the TPM_SFEINPUTS and TPM_SFEGATE need to be executed for each input wire and each input gate. Note that while the TPM processes the gates one-by-one, Flicker does not have the same restrictive memory constraints, so our Flicker module can take in up to 60 gates at a time and encrypt them. Table I shows the time taken for the TPM and Flicker to execute individual SFE commands.

TABLE I
SFE COMMAND TIMES

Command	TPM (seconds)	Flicker (seconds)
TPM_SFEInit	0.001 s	3 s
TPM_SFEInputs	0.003 s	4 s
TPM_SFEGate	0.05 s	5 s
TPM_SFEFinish	0.2 s	4 s

C. SFE Timings

The main purpose of creating the circuit in two ways was to compare the performance of the TPM and Flicker to decide which option was more feasible: extending the TPM to support SFE commands or use existing trusted computing infrastructure to support SFE. The time taken for creating and evaluating the TPM-assisted garbled circuit and Flicker-assisted garbled circuit are as shown in Table II. Since the overhead incurred by Flicker is around 3-4 seconds per command, and the times per command as executed by the TPM are around 0.004-0.05 seconds (excluding the TPM_SFEFinish command which is a one-time cost), the TPM outperforms Flicker by a decent margin. The reason why the difference between the TPM and Flicker timings is not more pronounced, in spite of the heavy overhead incurred by Flicker, is because we can pass in 60 gates at a time to Flicker while a TPM can process only one gate at a time. The evaluation times, which are shown in Table III, are the same as in a standard (non-hardware-assisted) SFE application since evaluation does not use any kind of hardware support. Note that evaluation times include the cost of oblivious transfer, which is the dominant cost.

TABLE II
HARDWARE-ASSISTED CIRCUIT CREATION TIME

Number of Gates	TPM (seconds)	Flicker (seconds)
501	25.3 s	56 s
1001	51.81 s	98 s
1501	74.11 s	137 s
2001	102.84 s	186 s

TABLE III
CIRCUIT EVALUATION TIME

Number of Gates	Evaluation time (seconds)
501	3 s
1001	5.6 s
1501	7.0 s
2001	8.9 s

VI. CONCLUSION AND FUTURE WORK

We have proposed a garbled circuit-based solution for secure function evaluation in the malicious model, and compared the performance using two different ways of applying trusted computing to the garbled circuit construction: using a modified TPM, and using a trusted computing infrastructure called *Flicker*. The TPM-based implementation is faster than the Flicker-based implementation, but cannot be performed without extensions to the current TPM 1.2 specifications.

For garbled circuit construction, we use Kolesnikov and Schneider's circuit construction technique, but without the use of "free-XOR gates," which would require an unbounded amount of storage for the XOR gate option. One direction for future work is to find a solution where we can use free-XOR gates with limited storage, such as the identity gates suggested by Järvinen *et al.* [11]. This would be useful since the circuit creator wouldn't have to compute and transmit extra truth tables to the evaluator. Another important direction for future work is to consider the more recent TrustVisor [32] system, which could potentially allow circuit construction — or even direct trusted computation on a small scale — with the same advantages as Flicker (i.e., no TPM extensions are required) but with significantly reduced overhead.

REFERENCES

[1] A. Yao, "How to generate and exchange secrets," in *Proc. of* 27^{th} *IEEE FOCS*, 1986, pp. 162–167.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC*, 2009, pp. 169–178.

[3] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proc. of 19th ACM STOC*, 1987, pp. 218–229.

[4] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *Proceedings of the twelfth annual ACM-SIAM SODA*, 2001, pp. 448–457.

[5] M. Naor, B. Pinkas, and R. Sumner, "Privacy preserving auctions and mechanism design," in *Proc. of* 1^{st} *ACM Conference on Electronic Commerce*, 1999, pp. 129–139.

[6] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth, "Cryptographic security for mobile code," in *Proc. of the IEEE Symposium on Security and Privacy*, 2001, pp. 2–11.

[7] S. R. Tate and K. Xu, "Mobile agent security through multi-agent cryptographic protocols," in *Proc. of the 4th International Conference on Internet Computing (IC 2003)*, 2003, pp. 462–468.

[8] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proc. of ACM STOC*, 1988, pp. 1–10.

[9] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *Advances in Cryptology - EUROCRYPT*, 2007, pp. 52–78.

[10] V. Goyal, P. Mohassel, and A. Smith, "Efficient two-party ad multi-party computation against covert adversaries," in *Advances in Cryptology - EUROCRYPT '08*, 2008, pp. 289–306.

[11] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider, "Embedded SFE: Offloading server and network using hardware tokens," in *Proc. of 14th FC*, 2010, pp. 207–221.

[12] A. Sadeghi, T. Schneider, and M. Winandy, "Token-based cloud computing: Secure outsourcing of data and arbitrary computations with lower latency," in *TRUST*, 2010, pp. 417–429.

[13] Trusted Computing Group, "Trusted Platform Module Specifications – Parts 1–3," Available at https://www.trustedcomputinggroup.org/specs/TPM/.

[14] V. Gunupudi and S. R. Tate, "Timing-accurate TPM simulation for what-if explorations in trusted computing," in *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2010, pp. 171–178.

[15] Trusted Computing Group, "Protection profile — PC client specific trusted platform module," Available at https://www.trustedcomputinggroup.org/specs/TPM/, tPM Family 1.2; Level 2.

[16] D. Chaum and T. P. Pedersen, "Wallet databases with observers (extended abstract)," in *Advances in Cryptology - CRYPTO*. Springer-Verlag, 1992, pp. 89–105.

[17] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Jounal of ACM*, pp. 431–473, 1996.

[18] J. Katz, "Universally composable multi party computation using tamper-proof hardware," in *EUROCRYPT*, 2007, pp. 115–128.

[19] C. Hazay and Y. Lindell, "Constructions of truly practical secure protocols using standard smartcards," in *ACM CCS*, 2008, pp. 491–500.

[20] S. Goldwasser, Y. Kalai, and G. Rothblum, "One-time programs," in *CRYPTO*, 2008, pp. 39–56.

[21] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tmaper-proof hardware tokens," in *TCC 2010*, 2010, pp. 308–326.

[22] A. Herzberg and H. Shulman, "Secure guaranteed computation," http://eprint.iacr.org/2010/449.

[23] V. Gunupudi and S. R. Tate, "Generalized non interactive oblivious transfer using count-limited objects with applications to secure mobile agents," in *FC*, 2008, pp. 98–112.

[24] S. Tate and R. Vishwanathan, "Improving cut-and-choose in verifiable encryption and fair exchange protocols using trusted computing technology," in *Proc. of the 23rd DBSec*, 2009, pp. 252–267.

[25] M. Fort, F. Freiling, L. Penso, Z. Benenson, and D. Kesdogan, "Trustedpals: secure multiparty computation implemented with smart cards," in *ESORICS*, 2006, pp. 34–48.

[26] A. Iliev and S. Smith, "Faerieplay on tiny trusted third parties," in *WATC*, 2006.

[27] ——, "Small, stupid and scalable: secure computing with faerieplay," in *STC workshop*, 2010, pp. 41–52.

[28] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free xor gates and applications," in *ICALP*, 2008, pp. 486–498.

[29] M. Bellare, "New proofs for NMAC and HMAC: Security without collision-resistance," in *CRYPTO*. Springer-Verlag, 2006, pp. 602–619.

[30] L. Sarmenta, M. van Dijk, C. O'Donnell, J. Rhodes, and S. Devadas, "Virtual monotonic counters and count-limited objects using a TPM without a trusted OS," in *STC*, 2006.

[31] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *EuroSys*, 2008, pp. 315–328.

[32] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: efficient TCB reduction and attestation," in *Proc. of IEEE Symposium on Security and Privacy*, 2010, pp. 143–158.

[33] T. Muller, L. Sarmenta, and J. Rhodes, "TPM/J - Java based API for the Trusted Platform Module TPM," Available at http://projects.csail.mit.edu/tc/tpmj/.

[34] M. Strasser, H. Stamer, and J. Molina, "Software-based TPM simulator," http://tpm-emulator.berlios.de.

[35] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. hua Teng, "On trip planning queries in spatial databases," in *SSTD*, 2005, pp. 273–290.