Embedded SFE: Offloading Server and Network Using Hardware Tokens

Kimmo Järvinen[1], Vladimir Kolesnikov[2], Ahmad-Reza Sadeghi[3], and Thomas Schneider[3]

[1] Dep. of Information and Comp. Science, Aalto University, Finland
kimmo.jarvinen@tkk.fi*

[2] Alcatel-Lucent Bell Laboratories, Murray Hill, NJ 07974, USA
kolesnikov@research.bell-labs.com

[3] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
{ahmad.sadeghi,thomas.schneider}@trust.rub.de**

Abstract. We consider Secure Function Evaluation (SFE) in the client-server setting where the server issues a secure token to the client. The token is not trusted by the client and is *not* a trusted third party.

We show how to take advantage of the token to drastically reduce the communication complexity of SFE and computation load of the server.

Our main contribution is the detailed consideration of design decisions, optimizations, and trade-offs, associated with the setting and its strict hardware requirements for practical deployment. In particular, we model the token as a computationally weak device with small constant-size memory and limit communication between client and server.

We consider semi-honest, covert, and malicious adversaries. We show the feasibility of our protocols based on a FPGA implementation.

1 Introduction

Secure and efficient evaluation of arbitrary functions on private inputs has been subject of cryptographic research for decades. In particular, the following scenario appears in a variety of practical applications: a service provider (server $\mathcal{S}$) and user (client $\mathcal{C}$) wish to compute a function f on their respective private data, without incurring the expense of a trusted third party. This can be solved interactively using Secure Function Evaluation (SFE) protocols, for example using the very efficient garbled circuit (GC) approach [26]. However, GC protocols potentially require a large amount of data to be transferred between $\mathcal{S}$ and $\mathcal{C}$. This is because f needs to be encrypted (garbled) as $\widetilde{f}$ and transferred from $\mathcal{S}$ to $\mathcal{C}$. In fact, the communication complexity of GC-based SFE protocols is dominated by the size of the GC, which can reach Megabytes or Gigabytes even for relatively small and simple functions (e.g., the GC for AES has size 0.5 MBytes [23]). Further, if security against more powerful adversaries is required, the use

* Supported by EU FP7 project CACE.
** Supported by EU FP6 project SPEED, EU FP7 project CACE and ECRYPT II.

of the standard cut-and-choose technique implies transfer of multiple GCs. (For covert adversaries, the transfer of only one GC is sufficient [7].)

While transmission of this large amount of data is possible for exceptional occurrences, in most cases, the network will not be able to sustain the resulting traffic. This holds especially for larger-scale deployment of secure computations, e.g., by banks or service providers, with a large number of customers. Additional obstacles include energy consumption required to transmit/receive the data, and the resulting reduced battery life in mobile clients, such as smartphones.[1]

Further, computational load on $\mathcal{S}$ (computing $\widetilde{f}$) is also a significant problem, especially in the case of large-scale deployment of SFE.

Our setting, goals and approach. Motivated by the possibility of large-scale and decentralized SFE deployment we aim to remove the expensive communication requirement, and to shift some of $\mathcal{S}$'s computation to $\mathcal{C}$. To this end, we note that in SFE protocols, and, in particular, in GC, the role of the server can be split between two entities, with the introduction of a new entity – secure token $\mathcal{T}$, which is placed in client $\mathcal{C}$'s possession, but executes $\mathcal{S}$'s code thus offloading $\mathcal{S}$. Further, it is possible to eliminate most of the communication between $\mathcal{C}$ and $\mathcal{S}$ and replace this with local communication between $\mathcal{C}$ and $\mathcal{T}$. A number of technical issues arises in this setting, which we address in this work.

More specifically, we discuss and analyze hardware-supported SFE, where the service provider $\mathcal{S}$ issues a *secure* (see below) hardware token $\mathcal{T}$ to $\mathcal{C}$. $\mathcal{C}$ communicates locally with $\mathcal{T}$, and remotely with $\mathcal{S}$. There is no direct channel between $\mathcal{T}$ and $\mathcal{S}$, but of course $\mathcal{C}$ can pass (and potentially interfere with) messages between $\mathcal{T}$ and $\mathcal{S}$. $\mathcal{T}$ is created by $\mathcal{S}$, so $\mathcal{S}$ trusts $\mathcal{T}$; however, as $\mathcal{C}$ does not trust $\mathcal{S}$, she also does not trust the token $\mathcal{T}$ to behave honestly.[2]

Attack model. We consider all three standard types of adversaries: semi-honest (follows protocol but analyzes the transcript), malicious (arbitrary behavior, cheating is always caught), and covert (cheating is caught with a certain deterrence probability, e.g., 1/2).

Hardware assumption. We assume $\mathcal{T}$ is tamper-proof or tamper-resistant. We argue that this assumption is reasonable. Indeed, while every token can likely be broken into, given sufficient resources, we are motivated by the scenarios where the payoff of the break is far below the cost of the break. This holds for relatively low-value transactions such as cell phone or TV service, where the potential benefit of the attack (e.g., free TV for one user) is not worth the investment of thousands or tens of thousands of dollars to break into the card. For higher-value applications one could raise the cost of the attack by using a high-end token $\mathcal{T}$, e.g., a smart card certified at FIPS 140-2, level 3 or 4.

[1] In some cases, the impact can be mitigated by creating and transferring GCs in the precomputation phase. However, this is not fully satisfactory. Firstly, even more data needs to be transferred since demand cannot be perfectly predicted. Further, this creates other problems, such as requiring large long-term storage on client devices.

[2] Note, if $\mathcal{C}$ in fact trusts $\mathcal{T}$ to behave honestly, then there exists a trivial solution, where $\mathcal{C}$ would let $\mathcal{T}$ compute the function on her inputs [11].

Hardware restrictions. As we assume the token to be produced in large quantities, we try to minimize its costs (e.g., chip surface) and make the assumptions on it as weak as possible. In particular our token requires only restricted computational capabilities (no public-key operations) and small constant secure RAM. We consider $\mathcal{T}$ with and without small constant secure non-volatile storage.

Envisioned Applications. As mentioned above, we aim to bring SFE closer to a large-scale deployment. The need to minimize communication forces us to rely on tokens, the issuance of which requires certain logistical capabilities. Therefore, we believe client-server applications are most likely to be the early adopters of SFE. Further, the natural trust model (semi-honest or covert server and malicious client) allow for efficient GC protocols. Finally, many client-server application scenarios naturally include financial and other transactions which involve sensitive, in particular privacy-sensitive, information.

Today, many service providers already distribute trusted tokens to their users. Examples include SIM cards in users' cell phones, and smart cards in users' TV set-top boxes. Bank- and credit cards often contain embedded secure chips. Special purpose (e.g., diagnostic) medical devices, which need to be monitored and controlled, are often issued to users at home. In these settings, it is natural to use the existing infrastructure to enhance the security and privacy guarantees of the offered products, and to offer new products previously impossible due to privacy violation concerns. We expand this discussion and consider other applications, such as privacy protection in targeted advertisement, content delivery, and remote medical diagnostics, in the full version [13].

1.1 Our Contributions and Outline

Our main contribution is architecture design, implementation, a number of optimizations, and detailed analysis of two-party SFE aided by a server-issued low-cost tamper-proof token. The communication complexity of our protocols is linear in the size of the input, and is independent of the size of the evaluated functionality. Further, most of the work of $\mathcal{S}$ can be offloaded to $\mathcal{T}$.

We use GC techniques of [15] and offer no-cost XOR gates. We rely on cheap hardware – the token $\mathcal{T}$ only executes symmetric-key operations (e.g., SHA and AES). $\mathcal{T}$ has small constant-size RAM (much smaller than the size of the circuit), but we do not resort to implementing expensive secure external RAM.

We provide two solutions; in one, $\mathcal{T}$ keeps state in secure non-volatile storage (a monotonic counter), while in the other, $\mathcal{T}$ maintains no long-term state.

We consider semi-honest, covert [7], and malicious [16] adversaries; our corresponding communication improvements are shown in Table 1.

Outline. We start with outlining our model and architecture in §3. We describe the protocols for both stateful and stateless $\mathcal{T}$, and state the security claim in §4. In §5, we discuss technical details of our FPGA prototype implementation, present timings and measurements, and show practicality of our solution.

Table 1. Communication between server $\mathcal{S}$ and client $\mathcal{C}$ for secure evaluation of function f with n inputs, statistical security parameter s, and deterrence probability $1-1/r$

Security	Previous Work	This Work
semi-honest	[26] $\mathcal{O}(\|f\| + n)$	$\mathcal{O}(n)$
covert	[7] $\mathcal{O}(\|f\| + sn + r)$	$\mathcal{O}(sn + r)$
malicious	[16] $\mathcal{O}(s\|f\| + s^2 n)$	$\mathcal{O}(s^2 n)$

1.2 Related Work

Related work on using tokens for secure computations falls in the following three categories, summarized in Table 2.

Table 2. Secure Protocols using Hardware Tokens. Columns denote the number of tokens, who trusts the token(s), if token(s) are stateful or stateless, and perform public-key operations. Properties more desired for practical applications in bold font.

Type	Reference	Functionality	# Tokens	Trusted by	Stateful	PK ops
A)	[10]	UC commitment	2	both	yes	yes
	[14,4]	UC commitment	2	**issuer**	yes	yes
	[3]	UC commitment	2	**issuer**	**no**	yes
	[19]	UC commitment	**1**	**issuer**	yes	**no**
B)	[9]	Set Intersection, ODBS	**1**	both	yes	**no**
	[8]	Non-Interact. OT	**1**	both	yes	yes
	[25]	Verif. Enc., Fair Exch.	**1**	both	yes	yes
C)	[6]	**SFE**	2	both	yes	yes
	[11]	**SFE**	**1**	both	yes	yes
	This Work	**SFE**	**1**	**issuer**	yes / **no**	**no**

A) Setup assumptions for the universal composability (UC) framework. As shown in [1], UC SFE protocols can be constructed from UC commitments. In turn, UC commitments can be constructed from signature cards trusted by both parties [10], or from tamper-proof tokens created and trusted only by the issuing party [14,19,3,4]. Here, [3] consider stateless tokens, and [19] require only one party to issue a token. This line of research mainly addresses the feasibility of UC computation based on tamper-proof hardware and relies on expensive primitives such as generic zero-knowledge proofs. Our protocols are far more practical.

B) Efficiency Improvements for Specific Functionalities. Efficient protocols with a tamper-proof token trusted by both players have been proposed for specific functionalities such as set intersection and oblivious database search (ODBS) [9], non-interactive oblivious transfer (OT) [8], and verifiable encryption and fair exchange [25]. In contrast, we solve the general SFE problem.

C) Efficiency Improvements for Arbitrary Functionalities. Clearly, SFE is efficient if aided by a trusted third party (TTP), who simply computes the function. SFE aided by hardware TTP was considered, e.g., in [6,11]. In contrast, we do not use TTP; our token is only trusted by its issuer.

2 Preliminaries

Notation. We denote symmetric security parameter by t (e.g., $t = 128$), and pseudo-random function (PRF) keyed with k and evaluated on x by $\mathsf{PRF}_k(x)$. PRF can be instantiated with a block cipher, e.g., AES, or a cryptographic hash function H, e.g., SHA-256, which we model as a Random Oracle (RO). AES is preferable if PRF is run repeatedly with same k as AES's key schedule amortizes. Message authentication code (MAC) keyed with k and evaluated on message m is denoted by $\mathsf{MAC}_k(m)$. We use a MAC that does not need to store the entire message, but can operate "online" on small blocks, e.g., AES-CMAC [24].

2.1 Garbled Circuits (GC)

Yao's Garbled Circuit approach [26] is the most efficient method for secure evaluation of a boolean circuit C. We summarize its ideas in the following. The circuit *constructor* (server $\mathcal{S}$) creates a *garbled circuit* $\widetilde{C}$: for each wire w_i of the circuit, he randomly chooses two garblings $\widetilde{w}_i^0, \widetilde{w}_i^1$, where $\widetilde{w}_i^j$ is the *garbled value* of w_i's value j. (Note: $\widetilde{w}_i^j$ does not reveal j.) Further, for each gate G_i, $\mathcal{S}$ creates a *garbled table* $\widetilde{T}_i$ with the following property: given a set of garbled values of G_i's inputs, $\widetilde{T}_i$ allows to recover the garbled value of the corresponding G_i's output, but nothing else. $\mathcal{S}$ sends these garbled tables, called *garbled circuit* $\widetilde{C}$ to the *evaluator* (client $\mathcal{C}$). Additionally, $\mathcal{C}$ obliviously obtains the *garbled inputs* $\widetilde{w}_i$ corresponding to inputs of both parties: the garbled inputs $\widetilde{y}$ corresponding to the inputs y of $\mathcal{S}$ are sent directly and $\widetilde{x}$ are obtained with a parallel 1-out-of-2 oblivious transfer (OT) protocol [20]. Now, $\mathcal{C}$ can evaluate the garbled circuit $\widetilde{C}$ on the garbled inputs to obtain the *garbled outputs* by evaluating $\widetilde{C}$ gate by gate, using the garbled tables $\widetilde{T}_i$. Finally, $\mathcal{C}$ determines the plain values corresponding to the obtained garbled output values using an output translation table received by $\mathcal{S}$. Correctness of GC follows from the way garbled tables $\widetilde{T}_i$ are constructed.

Improved Garbled Circuit with free XOR [15]. An efficient method for creating garbled circuits which allows "free" evaluation of XOR gates was presented in [15]. More specifically, a garbled XOR gate has no garbled table (*no communication*) and its evaluation consists of XORing the its garbled input values (*negligible computation*) – details below. The other gates, called *non-XOR gates*, are evaluated as in Yao's GC construction [26] with a *point-and-permute technique* (as used in [18]): The garbled values $\widetilde{w}_i = \langle k_i, \pi_i \rangle \in \{0,1\}^{t'}$ consist of a symmetric key $k_i \in \{0,1\}^t$ and a random permutation bit $\pi_i \in \{0,1\}$ (recall, t is the symmetric security parameter). The entries of the garbled table are permuted such that the permutation bits π_i of a gate's garbled input wires can be used as index into the garbled table to directly point to the entry to be decrypted. After decrypting this entry using the garbled input wires' t-bit keys k_i, evaluator obtains the garbled output value of the gate. The encryption is done with the symmetric encryption function $\mathsf{Enc}^s_{k_1,\ldots,k_d}(m)$, where d is the number of inputs of the gate and s is a unique identifier used once. Enc can be instantiated with $m \oplus \mathsf{H}(k_1||\ldots||k_d||s)$, where H is a RO. We note that the RO assumption can be avoided or weakened at small additional computation cost – see full version [13]. The main observation of [15] is, that the constructor $\mathcal{S}$ chooses a global key difference $\Delta \in_R \{0,1\}^t$ which remains unknown to evaluator $\mathcal{C}$ and relates the garbled values as $k_i^0 = k_i^1 \oplus \Delta$. Clearly, the usage of such garbled values allows for *free evaluation of XOR gates* with input wires w_1, w_2 and output wire w_3 by computing $\widetilde{w}_3 = \widetilde{w}_1 \oplus \widetilde{w}_2$ (no communication and negligible computation).

3 Architecture, System and Trust Model

We present in detail our setting, players, and hardware and trust assumptions.

As shown in Fig. 1, there are three parties – client $\mathcal{C}$, server $\mathcal{S}$ and tamper-resistant token $\mathcal{T}$, issued and trusted by $\mathcal{S}$. Our goal is to let $\mathcal{C}$ and $\mathcal{S}$ securely evaluate a public function f on their respective private inputs x and y.

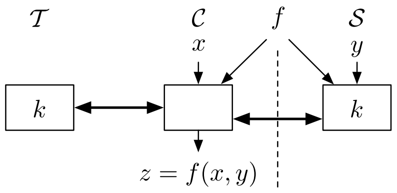


Fig. 1. Model Overview

Communication. $\mathcal{C} \leftrightarrow \mathcal{S}$: We view this as an expensive channel. Communication $\mathcal{C} \leftrightarrow \mathcal{S}$ flows over the Internet, and may include a wireless or cellular link. This implies small link bandwidth and power consumption concerns of mobile devices. We wish to minimize the utilization of this channel.

$\mathcal{T} \leftrightarrow \mathcal{C}$: As $\mathcal{T}$ is held locally by $\mathcal{C}$, this is a cheap channel (both in terms of bandwidth and power consumption), suitable for transmission of data linear in the size of f, or even greater.

$\mathcal{T} \leftrightarrow \mathcal{S}$: There is no direct channel between $\mathcal{T}$ and $\mathcal{S}$, but, of course, $\mathcal{C}$ can pass (and potentially interfere with) messages between $\mathcal{T}$ and $\mathcal{S}$.

Trust. $\mathcal{C} \leftrightarrow \mathcal{S}$: As in the standard SFE scenario, $\mathcal{C}$ and $\mathcal{S}$ don't trust each other. We address semi-honest, covert, and malicious $\mathcal{C}$ and $\mathcal{S}$.

$\mathcal{S} \leftrightarrow \mathcal{T}$: $\mathcal{T}$ is fully trusted by $\mathcal{S}$, since $\mathcal{T}$ is tamper-resistant. $\mathcal{S}$ and $\mathcal{T}$ share a secret key k, used to establish a secure channel and to derive joint randomness.

$\mathcal{T} \leftrightarrow \mathcal{C}$: $\mathcal{C}$ does not trust $\mathcal{T}$, as $\mathcal{T}$ is the agent of $\mathcal{S}$, and may communicate with $\mathcal{S}$ through covert channels.

Storage, computation and execution. $\mathcal{C}$ and $\mathcal{S}$ are computationally strong devices which can perform both symmetric- and asymmetric-key operations.[3] Both have sufficient memory, linear in the size of f. $\mathcal{C}$ has control over $\mathcal{T}$, and can reset it, e.g., by interrupting its power supply. As justified in §1, $\mathcal{T}$ is a cheap

[3] If needed, $\mathcal{C}$'s capabilities may be enhanced by using a trusted hardware accelerator.

special purpose hardware with minimum chip surface: $\mathcal{T}$ has circuitry only for evaluating symmetric-key primitives in hardware (no public-key or true random number generator) and has a small secure RAM. It may (§4.3) or may not (§4.4) have small non-volatile secure storage[4], unaffected by the resets by $\mathcal{C}$.

4 Token-Assisted Garbled Circuit Protocols

In our presentation, we assume reader's familiarity with the GC technique, including free XORs of [15] (cf. §2.1), and concentrate on the aspects specific to the token setting. We start with a high-level description of our protocol. Then, in §4.2 - §4.4, we present the technical details of our construction – efficient circuit representation, and GC generation by stateful and stateless tokens.

4.1 Protocols Overview, Security Intuition and Claim

Our constructions are a natural (but technically involved) modification of standard GC protocols, so as to split the actions of the server into two parts – now executed by $\mathcal{S}$ and $\mathcal{T}$ – while maintaining provable security. We offload most of the work (notably, GC generation and output) to $\mathcal{T}$, thus achieving important communication savings, and partially offloading $\mathcal{S}$'s computation to $\mathcal{T}$.

We start our discussion with the solution in the semi-honest model. However, our modification of the basic GC is secure against malicious actions, and our protocols are easily and efficiently extendible to covert and malicious settings.

At the high level, our protocols work as shown in Fig. 2: $\mathcal{C}$ obtains the garbled inputs $\widetilde{x}, \widetilde{y}$ from $\mathcal{S}$, and the garbled circuit $\widetilde{f}$ corresponding to the function f from $\mathcal{T}$. Then, $\mathcal{C}$ evaluates $\widetilde{f}$ on $\widetilde{x}, \widetilde{y}$ and obtains the result $z = f(x, y)$.

It is easy to see that the introduction of $\mathcal{T}$ and offloading to it some of the computation does not strengthen $\mathcal{S}$, and thus does not bring security concerns for $\mathcal{C}$ (as compared to standard two-party GC). On the other hand, separating

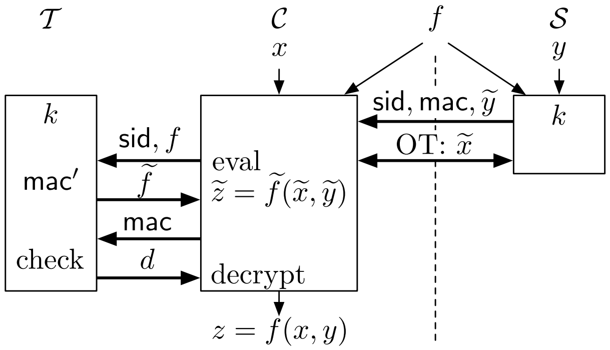


Fig. 2. Protocols Overview

[4] $\mathcal{T}$'s key k is a fixed part of its circuit, and is kept even without non-volatile storage.

the states of $\mathcal{S}$ and $\mathcal{T}$, placing $\mathcal{C}$ in control of their communication, and $\mathcal{C}$'s ability to reset $\mathcal{T}$ introduces attack opportunities for $\mathcal{C}$. We show how to address these issues with the proper synchronization and checks performed by $\mathcal{S}$ and $\mathcal{T}$.

Our main tool is the use of a unique session id sid for each GC evaluation. From sid and the shared secret key k, $\mathcal{S}$ and $\mathcal{T}$ securely derive a session key K, which is then used to derive the randomness used in GC generation. Jumping ahead (details in §4.3), we note that sid uniqueness is easily achieved if $\mathcal{T}$ is stateful simply by setting sid equal to the value of the strictly monotonic session counter ctr maintained by $\mathcal{T}$. However, if $\mathcal{T}$ is stateless, $\mathcal{C}$ can always replay $\mathcal{S}$'s messages. In §4.4 we show how to ensure that replays do not help $\mathcal{C}$.

Since $\mathcal{S}$ and $\mathcal{T}$ derive the same randomness for each session, the (same) garbled circuit $\widetilde{f}$ can be generated by $\mathcal{T}$. Unfortunately, the weak $\mathcal{T}$ cannot store the entire f. Instead, $\mathcal{C}$ provides the circuit corresponding to function f gate-by-gate to $\mathcal{T}$, and obtains the corresponding garbled gate of $\widetilde{f}$. The garbled gate can immediately be evaluated by $\mathcal{C}$ and needs not to be stored. $\mathcal{C}$ is prevented from providing a wrong f to $\mathcal{T}$, as follows. First, $\mathcal{S}$ issues a MAC of f, e.g., $\mathsf{mac} = \mathsf{MAC}_k(\mathsf{sid}, f)$, where f is the agreed circuit representation of the evaluated function (cf. §4.2). Further, $\mathcal{T}$ computes its version of the above MAC, mac', as it answers $\mathcal{C}$'s queries in computing $\widetilde{f}$. Finally, $\mathcal{T}$ reveals the decryption information d that allows $\mathcal{C}$ to decrypt the output wires only if $\mathcal{C}$ provides the matching mac.

Garbled Inputs. The garbled input $\widetilde{y}$ of $\mathcal{S}$ can be computed by $\mathcal{S}$ and sent to $\mathcal{C}$, requiring $|y| \cdot t$ bits communication, where t is the security parameter. Alternatively, if $\mathcal{T}$ is stateful, $\mathcal{S}$ can establish a secure channel with $\mathcal{T}$, e.g., based on session key K, send y over the channel, and have $\mathcal{T}$ output $\widetilde{y}$ to $\mathcal{C}$. This achieves the optimal communication between $\mathcal{S}$ and $\mathcal{C}$ of $|y|$ bits.

The garbling $\widetilde{x}$ of $\mathcal{C}$'s input can be transferred from $\mathcal{S}$ to $\mathcal{C}$ with a parallel OT protocol which requires $\mathcal{O}(|x|t)$ bits of communication. Alternatively, the efficient OT extension of [12] which reduces many OTs to a small number of OTs (depending on security parameter t) can be adopted to our token-based scenario as described in the full version [13, §C]. This reduces the communication between $\mathcal{S}$ and $\mathcal{C}$ to $\mathcal{O}(t^2)$ which is independent of the size of the input x.

Extension to Covert and Malicious Parties. Standard GC protocols for covert [7] or malicious [16] adversaries rely on the following cut-and-choose technique. $\mathcal{S}$ creates multiple GCs $\widetilde{C}_i$, deterministically derived from random seeds s_i, and commits to each, e.g., by sending $\widetilde{C}_i$ or $\mathsf{H}(\widetilde{C}_i)$ to $\mathcal{C}$. In covert case, $\mathcal{C}$ asks $\mathcal{S}$ to open all but one garbled circuit I by revealing the corresponding $s_{i \neq I}$. For all opened circuits, $\mathcal{C}$ computes $\widetilde{C}_i$ and checks that they match the commitments. The malicious case is similar, but $\mathcal{C}$ asks $\mathcal{S}$ to open half of the circuits, evaluates the remaining ones and chooses the majority of their results.

These protocols similarly benefit from our token-based separation of the server into $\mathcal{S}$ and $\mathcal{T}$. As in the semi-honest protocol, the GC generation can be naturally offloaded to $\mathcal{T}$, achieving corresponding computation and communication relief on the server and network resources. GC correctness verification is achieved by requesting $\mathcal{S}$ to reveal the generator seeds $s_{i \neq I}$. (Of course, these "opened" circuits are not evaluated.) Note that requirements on $\mathcal{T}$ are the same as in the semi-honest setting. Further, in both covert and malicious cases, the communication between $\mathcal{C}$ and $\mathcal{S}$ is independent of the size of f. The resulting communication complexity of these protocols is summarized in Table 1.

Security Claim. For the lack of space, in this work we present our protocols implicitly, by describing the modifications to the base protocols of [15]. We informally argue the security of the modifications as they are described. Formal proofs can be naturally built from proofs of [15] and our security arguments. At the very high level, security against $\mathcal{S}/\mathcal{T}$ follows from the underlying GC protocols, since $\mathcal{S}$ is not stronger here than in the two-party SFE setting. The additional power of $\mathcal{C}$ to control the channel between $\mathcal{S}$ and stateful $\mathcal{T}$ is negated by establishing a secure channel (§4.3). $\mathcal{C}$'s power to reset stateless $\mathcal{T}$ is addressed by ensuring that by replaying old messages $\mathcal{C}$ gets either what he already knows, or completely unrelated data (§4.4).

Theorem 1. *Assuming $\mathcal{T}$ is tamper-proof, protocols described throughout this section are secure in the semi-honest, covert, and malicious models respectively.*

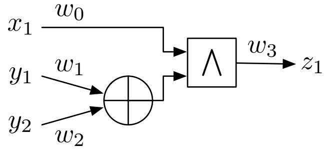
4.2 Circuit Representation

We now describe our circuit representation format. Our criteria are compactness, the ability to accommodate free XOR gates of [15], and ability of $\mathcal{T}$ to process the encoding "online", i.e., with small constant memory. Recall, our $\mathcal{T}$ operates in request-response fashion. $\mathcal{C}$ incrementally, gate-by-gate, "feeds" the circuit description to $\mathcal{T}$ which responds with the corresponding garbled tables.

We consider circuits with two-input boolean gates. We note that our techniques can be naturally generalized to general circuits.

Our format is derived from standard representations, such as that of Fairplay [18], with the necessary changes to support our requirements. For readability, we describe the format using a simple example circuit shown in Fig. 3. This circuit computes $z_1 = x_1 \wedge (y_1 \oplus y_2)$, where x_1 is the input bit of $\mathcal{C}$ and y_1, y_2 are two input bits of $\mathcal{S}$. The corresponding circuit representation shown on the right is composed from the description of the inputs, gates, and outputs as follows.

Inputs and wires: The wires w_i of the circuit are labeled with their index $i = \{0, 1, ...\}$ (wires with a larger fan-out are viewed as a single wire). The first X wires are associated with the input of $\mathcal{C}$, the following Y wires are associated with the input of $\mathcal{S}$, and the internal wires are labeled in topological order starting



Circuit Representation:
```
1 2
3 [0] [1 2] [0001]
3
```

Fig. 3. Example for Circuit Representation

from index $X + Y$ (output wires of XOR gates are not labeled, as XOR gates are incorporated into their successor gates as described in the next paragraph). The first line of the circuit description specifies X and Y (Fig. 3: $X = 1, Y = 2$).

Gates are labeled with the index of their outgoing wire; each gate description specifies its input wires. XOR gates do not have gate tables and are omitted from the description. Rather, non-XOR gates, instead of pointing to two input wires, include two input wire *lists*. If the input list contains more than one wire, these wire values are to be XORed to obtain the corresponding gate input. Gate's description concludes with its truth table. In Fig. 3, the second line describes the AND gate, which has index 3, and inputs w_0 and $w_1 \oplus w_2$.

Outputs: The circuit description concludes with Z lines which contain the indices of the Z output wires (Fig. 3: the only ($Z = 1$) output wire is w_3).

Large XOR sub-circuits. In this representation, XOR gates with fan-out > 1 occur multiple times in the description of their successor gates. In the worst case, this results in a quadratic increase of the circuit description. To avoid this cost, we insert an identity gate after each XOR gate with a large fan-out.

4.3 GC Creation with Stateful Token Using Secure Counter

The main idea of our small-RAM-footprint GC generation is having $\mathcal{T}$ generate garbled tables "on the fly". This is possible, since each garbled table can be generated only given the garblings of input and output wires. In our implementation, we pseudorandomly derive the wire garbling from the session key and wire index. The rest of this section contains relevant details.

Session Initialization. SFE proceeds in sessions, where one session is used to securely evaluate a function once. $\mathcal{T}$ has a secure monotonic session counter ctr which is (irreversibly) incremented at the beginning of each session. The session id sid is set to the incremented state of ctr. (We omit the discussion of synchronization of ctr between $\mathcal{T}$ and $\mathcal{S}$ which may happen due to communication and other errors.) Then, the session key is computed by $\mathcal{S}$ and $\mathcal{T}$ as $K = \mathsf{PRF}_k(\mathsf{sid})$ and subsequently used to provide fresh randomness to create the GC.

As required by the construction of [15] (cf. §2.1), the two garbled values of the same wire differ by a global difference offset Δ. This offset is derived from K at session initialization and kept in RAM throughout the session.

Subsequently, garbled wire values w_i are derived on-the-fly from K as

$$\widetilde{w}_i^0 = \mathsf{PRF}_K(i), \qquad \widetilde{w}_i^1 = \widetilde{w}_i^0 \oplus \Delta. \tag{1}$$

Garbled Gates. $\mathcal{T}$ receives the description of the circuit, line by line, in the format described in §4.2, and generates and outputs to $\mathcal{C}$ corresponding garbled gates, using only small constant memory. $\mathcal{T}$ first verifies that the gate with the same label had not been processed before. (Otherwise, by submitting multiple gate tables for the same gate, $\mathcal{C}$ may learn the real wire values). This is achieved by keeping the monotonically increasing processed gate counter gctr, verifying that gate's label $\mathsf{glabel} > \mathsf{gctr}$, and setting $\mathsf{gctr} = \mathsf{glabel}$. $\mathcal{T}$ then derives and stores garblings of the gate's input and output wires according to (1). (For input lists, the wire's garbling $\widetilde{w}^0$ is computed as the XOR of garblings of the listed wires, and $\widetilde{w}^1$ is set to $\widetilde{w}^0 \oplus \Delta$. Note that this requires constant RAM.) Finally, based on these garblings, gate's garbled table is computed and output to $\mathcal{C}$.

Garbled Outputs. Recall, $\mathcal{T}$ must verify circuit correctness by checking mac generated by $\mathcal{S}$. Thus, $\mathcal{T}$ does not release the output decryption tables to $\mathcal{C}$ until after the successful check. At the same time, the check is not complete until the entire circuit had been fed to $\mathcal{T}$. To avoid having $\mathcal{T}$ store the output decryption tables or involving $\mathcal{S}$ at this stage, $\mathcal{T}$ simply encrypts the output tables using a fresh key K', and outputs the key only upon a successful MAC verification.

4.4 GC Creation with Stateless Token (no Counter)

As discussed above, while non-volatile secure storage (the counter ctr) is essential in our protocol of §4.3, in some cases, it may be desired to avoid its cost. We now discuss the protocol amendments required to maintain security of SFE with the support of a token whose state can be reset by, e.g., a power interruption.

First, we observe that $\mathcal{S}$ is still able to maintain state, and choose unique counters. However, $\mathcal{T}$ can no longer be assured that sid claimed by $\mathcal{C}$ is indeed fresh. Further, $\mathcal{T}$ does not have a source of independent randomness, and thus cannot establish a secure channel with $\mathcal{S}$, e.g., by running a key exchange.

We begin with briefly describing a replay vulnerability of our protocol of §4.3, when $\mathcal{T}$ is executed with same sid. First, $\mathcal{C}$ properly executes SFE. Second time he runs $\mathcal{T}$ with the same sid, but feeds $\mathcal{T}$ an incorrect circuit, receiving valid garbled tables for each of the gates, generated for the *same* wire garblings. Now, even though $\mathcal{T}$ will not accept mac and will not decrypt the output wires, $\mathcal{C}$ had already received them in the first execution. It is easy to see that $\mathcal{C}$ "wins".

Our solution is to ensure that $\mathcal{C}$ does not benefit from replaying $\mathcal{T}$ with the same sid. To achieve this, we require that each wire garblings are derived from the (hash of the) entire gate description (i.e., id, truth table, and list of inputs), as described below. If $\mathcal{C}$ replays and gives a different gate description, she will not be able to relate the produced garbled table with a previous output of $\mathcal{T}$.

We associate with each wire w_i a (revealed to $\mathcal{C}$) hash value h_i. For input wires, h_i is the empty string. For each other wire i, h_i is derived (e.g., via Random Oracle) from the description of the gate i (which includes index, truth table, and list of inputs; cf. §4.2) that emits that wire: $h_i = \mathsf{H}(\langle gate\_description \rangle)$. The garbled value of wire w_i now depends on its hash value h_i: $\widetilde{w}_i^0 = \mathsf{PRF}_K(h_i)$ and $\widetilde{w}_i^1 = \widetilde{w}_i^0 \oplus \Delta$. Finally, to enable the computation of the garbled tables, $\mathcal{C}$ must feed back to $\mathcal{T}$ the hashes h_i of the input wires, and receive from $\mathcal{T}$ and keep for future use the hash of the output wire. As noted above, $\mathcal{C}$'s attempts to feed incorrect values result in the output of garbled tables that are unrelated to previous outputs of $\mathcal{T}$, and thus do not help $\mathcal{C}$.

5 Proof-of-Concept Implementation

We have designed a proof-of-concept implementation to show the practicability of our token-assisted GC protocols of §4. In the following we describe our architecture for the stateful token case of §4.3. Extension to the stateless case is straightforward. We instantiate PRF with AES-128, H and H with SHA-256, and MAC with AES-CMAC.

5.1 Architecture

Fig. 4 depicts the high level architecture of our design consisting of a two-stage pipeline and a MAC core. Stage 1 of the pipeline creates the garbled input and output values of a gate using an AES core and stage 2 computes the garbled table with a SHA core. The two-stage pipeline increases performance as two gates can be processed concurrently. The MAC core computes the authentication message mac' of the circuit provided by $\mathcal{C}$ (cf. §4.1).

Design Principle. To achieve maximum speed with minimum hardware resources, we followed a general guideline exploiting parallelism as long as it can be done without using several instances of the same algorithm. For example, we opted to compute the four entries of a garbled table with a single SHA core instead of using four parallel SHA cores which would have increased performance, but only with a significant increase in area. As the only exception, we included a separate

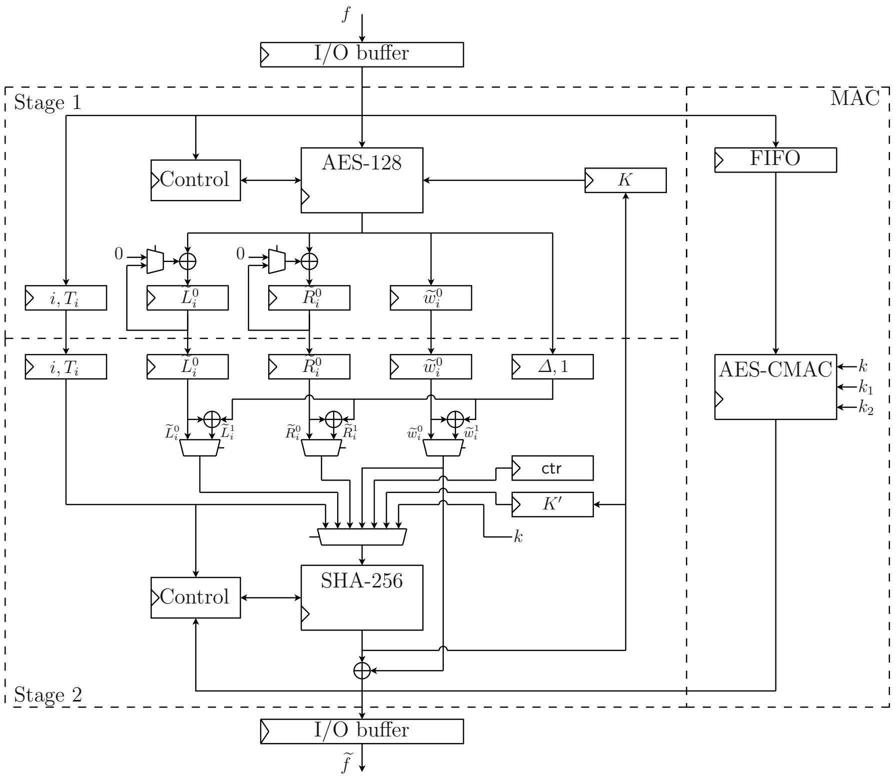


Fig. 4. Simplified architectural diagram of our proof-of-concept implementation. Selectors of multiplexers and write enables of registers are set by additional control logics.

MAC core rather than reusing the AES core of stage 1, because it would have severely complicated the control of the pipeline.

Description of Operation. In the session initialization (cf. §4.4), the SHA core in stage 2 derives session key K and output encryption key K' from key k and current counter value $\mathsf{ctr} = \mathsf{sid}$ which is used as key for the AES core. Then, the key difference Δ is derived with the AES core and stored in a register. The circuit is provided gate-by-gate into the input buffer in the format described in §4.2 (gate table denoted by T_i in Fig. 4). Stage 1 starts to process a gate by deriving the garbled output value $\widetilde{w}_i^0$. Then, the two garbled inputs of the gate ($\widetilde{L}_i^0, \widetilde{R}_i^1$ in Fig. 4) are derived by XORing the garblings listed in the input wire lists one-by-one (see §4.3). When all garblings are derived they are forwarded to stage 2 and stage 1 processes the next gate. Stage 2 computes the garbled table and the encrypted output decryption tables and writes them into the output buffer. The MAC core operates independently from the two-stage pipeline.

5.2 Prototype Implementation

We implemented our architecture in VHDL.

Implementation Details. For the AES core we chose an iterative design of AES-128 with a latency of 10 clock cycles per encryption. The core includes an online key scheduling unit. The S-boxes are implemented as suggested in [2]; otherwise, the core is a straightforward implementation of the standard [21]. The SHA core implements SHA-256 with a latency of 67 clock cycles per 512-bit block. The core is a straightforward iterative design of the standard [22]. The MAC core includes an AES core implemented as above; otherwise the core is a straightforward implementation of AES-CMAC [24]. As the subkeys, k_1 and k_2, depend only on the key k they were precomputed and hardwired in the design.

FPGAs. We compiled the VHDL code for a low-end FPGA, the Altera Cyclone II EP2C20F484C7 FPGA, with Quartus II, version 8.1 (2008). We emphasize that this FPGA is for prototyping only, as it lacks secure embedded non-volatile memory for storing ctr (e.g., the Xilinx Spartan-3AN FPGAs has integrated Flash memory for this). The resulting area and memory requirements are listed in Table 3. The design occupies 60 % of logic cells and 21 % of memory blocks available on the device and runs at 66 MHz (the critical path for clock frequency is in the AES core). These results show that the design is, indeed, feasible for a low-cost implementation, for example, with low-cost FPGAs which, in turn, is mandatory for the practicability of the token-based scheme.
Smart Cards. In particular, we note that the requirements are sufficiently low also for contemporary smart card technologies, because AES-128 and SHA-256 require only about 3,400 and 11,000 gates, respectively [5]. As our protocol requires no public-key operations on the token, small smart cards are sufficient.

Performance. We determined the latency of our implementation with ModelSim, version 6.3g (2008). Overall, the latency is given as $\#\mathsf{clock\_cycles} = 158G_1 + 312G_2 + 154O + 150$, where G_1, G_2 is the number of 1-input gates respectively

Table 3. Results on an Altera Cyclone II FPGA (the hierarchy is as shown in Fig. 4)

Entity	Stage 1	Stage 2	MAC	IO	Total
Area (Logic cells)	3317 (30 %)	4539 (40 %)	3059 (27 %)	263 (3 %)	11231 (60 %)
Memory (M4K)	0 (0 %)	8 (73 %)	1 (9 %)	2 (18 %)	11 (21 %)

2-input gates and O is the number of outputs, assuming that each gate has at most 21 inputs in its input lists (if more, stage 2 needs to wait for stage 1) and that data I/O does not introduce additional delays.

Example 1. Our implementation generates a GC for 16-bit comparison ($G_1 = 0, G_2 = 16, O = 1$) in 5,296 clock cycles ($\approx$80 μs with 66 MHz clock). In Software, this takes roughly 0.5 s on an Intel Core 2 6420 at 2.13 GHz [17].

Example 2. Generating a GC for AES-128 encryption ($G_1 = 12614$, $G_2 = 11334$, $O = 128$) takes 5,549,082 clock cycles ($\approx$84 ms with 66 MHz clock). In Software, this takes approximately 1 s on an Intel Core 2 Duo at 3.0 GHz [23].

Acknowledgements. We would like to thank Wilko Henecka for preparing test circuits, and Ivan Damgård and reviewers of FC'10 for their helpful comments.

References

1. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally composable two-party and multi-party secure computation. In: STOC 2002, pp. 494–503 (2002)
2. Canright, D.: A very compact S-box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
3. Chandran, N., Goyal, V., Sahai, A.: New constructions for UC secure computation using tamper-proof hardware. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 545–562. Springer, Heidelberg (2008)
4. Damgård, I., Nielsen, J.B., Wichs, D.: Universally composable multiparty computation with partially isolated parties. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 315–331. Springer, Heidelberg (2009)
5. Feldhofer, M., Wolkerstorfer, J.: Strong crypto for RFID tags — a comparison of low-power hardware implementations. In: IEEE Symp. Circuits and Systems (ISCAS 2007), pp. 1839–1842 (2007)
6. Fort, M., Freiling, F.C., Penso, L.D., Benenson, Z., Kesdogan, D.: Trustedpals: Secure multiparty computation implemented with smart cards. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 34–48. Springer, Heidelberg (2006)
7. Goyal, V., Mohassel, P., Smith, A.: Efficient two party and multi party computation against covert adversaries. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 289–306. Springer, Heidelberg (2008)
8. Gunupudi, V., Tate, S.: Generalized non-interactive oblivious transfer using count-limited objects with applications to secure mobile agents. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 98–112. Springer, Heidelberg (2008)
9. Hazay, C., Lindell, Y.: Constructions of truly practical secure protocols using standard smartcards. In: Proc. ACM CCS, pp. 491–500. ACM, New York (2008)

10. Hofheinz, D., Müller-Quade, J., Unruh, D.: Universally composable zero-knowledge arguments and commitments from signature cards. In: Central European Conference on Cryptology (MoraviaCrypt 2005) (2005)
11. Iliev, A., Smith, S.: More efficient secure function evaluation using tiny trusted third parties. Technical Report TR2005-551, Dartmouth College, Computer Science, Hanover, NH (July 2005)
12. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003)
13. Järvinen, K., Kolesnikov, V., Sadeghi, A.-R., Schneider, T.: Embedded SFE: Offloading server and network using hardware tokens. In: Cryptology ePrint Archive, Report 2009/591 (2009), `http://eprint.iacr.org/`
14. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 115–128. Springer, Heidelberg (2007)
15. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 486–498. Springer, Heidelberg (2008)
16. Lindell, Y., Pinkas, B.: An efficient protocol for secure two-party computation in the presence of malicious adversaries. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 52–78. Springer, Heidelberg (2007)
17. Lindell, Y., Pinkas, B., Smart, N.: Implementing two-party computation efficiently with security against malicious adversaries. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 2–20. Springer, Heidelberg (2008)
18. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: USENIX Security Symposium 2004. USENIX Association (2004)
19. Moran, T., Segev, G.: David and goliath commitments: UC computation for asymmetric parties using tamper-proof hardware. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 527–544. Springer, Heidelberg (2008)
20. Naor, M., Pinkas, B.: Efficient oblivious transfer protocols. In: ACM-SIAM Symposium on Discrete Algorithms (SODA 2001), pp. 448–457. SIAM, Philadelphia (2001)
21. NIST, U.S. National Institute of Standards and Technology. Federal Information Processing Standards (FIPS 197). Advanced Encryption Standard (AES) (November 2001), `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
22. NIST, U.S. National Institute of Standards and Technology. Federal Information Processing Standards (FIPS 180-2). Announcing the Secure Hash Standard (August 2002), `http://csrc.nist.gov/publications/fips/fips180-2/fips-180-2.pdf`
23. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer, Heidelberg (2009)
24. Song, J.H., Poovendran, R., Lee, J., Iwata, T.: The AES-CMAC Algorithm. RFC 4493 (Informational) (June 2006), `http://tools.ietf.org/html/rfc4493`
25. Tate, S., Vishwanathan, R.: Improving cut-and-choose in verifiable encryption and fair exchange protocols using trusted computing technology. In: Gudes, E., Vaidya, J. (eds.) Data and Applications Security XXIII. LNCS, vol. 5645, pp. 252–267. Springer, Heidelberg (2009)
26. Yao, A.C.: How to generate and exchange secrets. In: IEEE Symposium on Foundations of Computer Science (FOCS 1986), pp. 162–167. IEEE, Los Alamitos (1986)