More Efficient Secure Function Evaluation Using Tiny Trusted Third Parties

Dartmouth Computer Science Technical Report TR2005-551

Alexander Iliev
sasho@cs.dartmouth.edu

Sean Smith
sws@cs.dartmouth.edu

July 2005

Abstract

We investigate the use of trustworthy devices, which function as trusted third parties (TTPs), to solve general two-party Secure Function Evaluation (SFE) problems. We assume that a really trustworthy TTP device will have very limited protected memory and computation environment—a *tiny TTP*. This precludes trivial solutions like "just run the function in the TTP".

Traditional scrambled circuit evaluation approaches to SFE have a very high overhead in using indirectly-addressed arrays—every array access's cost is linear in the array size. The main gain in our approach is that array access can be provided with much smaller overhead—$O(\sqrt{N}\log N)$. This expands the horizon of problems which can be efficiently solved using SFE. Additionally, our technique provides a simple way to deploy arbitrary programs on tiny TTPs.

In our prototype, we use a larger (and expensive) device, the IBM 4758 secure coprocessor, but we also speculate on the design of future tiny devices that could greatly improve the current prototype's efficiency by being optimized for the operations prevalent in our algorithms.

We have prototyped a compiler for the secure function definition language (SFDL) developed in the Fairplay project. Our compiler produces an arithmetic circuit, augmented with *array access gates* which provide more efficient secure access to arrays. We then have a circuit interpreter in the 4758 to evaluate such a circuit on given inputs. It does this gate by gate, requiring very little protected space. We report on the performance of this prototype, which confirms our approach's strength in handling indirectly-addressed arrays.

1 Introduction

The Secure Function Evaluation (SFE) problem asks how two (or N) parties can jointly compute an N-input and N-output function $f(x_1, x_2, \ldots, x_N)$ on their N respective inputs without any party learning more than what they can learn from their own partial result of the function.

The security and privacy conditions for SFE are usually stated in terms of an ideal Trusted Third Party (TTP): any correct protocol for SFE should behave as if Agnes and Boris had sent

their inputs to the TTP, the TTP had then computed the results, and distributed them to their respective recipients.

Motivated by the assumption that no sufficiently trustworthy TTPs exist, the actual solutions developed for the problem work without a TTP—they are protocols involving only the original players. These solutions are mostly based on evaluation of a *blinded circuit* computing the function f. An actual prototype of a circuit-based SFE system has recently been built–Fairplay [21]. Fairplay provides a compiler from a C-like high-level imperative language to a circuit format, and a runtime engine for two players to evaluate such a circuit securely on their inputs. Such circuit-based solutions for SFE incur a communication and computation cost linear in the circuit size, and thus the circuit size is the primary performance metric.

Hardware-based TTPs On the other hand, devices intended to function as actually trustworthy third parties have been introduced [27]. *Secure coprocessors* [34], realized commercially as products like the IBM 4758 [9, 28, 29], offer programmable general-purpose computing environments and memory, protected against malicious attack, even by an adversary with physical access to the device.

A possible solution to the SFE problem is for Agnes and Boris to submit their inputs to such a secure coprocessor *(SCop)*, which they can verify is running a program that correctly and securely implements f. The SCop does the computation and distributes the answers, and the relying parties have assurance (such as the IBM 4758's FIPS 140-1 Level 4 validation) that even the operator of the SCop cannot subvert the desired security and privacy properties[1].

In practice, however, using a hardware-based TTP for SFE requires overcoming two main obstacles:

- The need for physical security forces limits on the memory size and computational power of the device. How can we use a hardware-based TTP for secure function evaluation if the function doesn't fit inside the device? Having the TTP use external resources reveals information to the adversary lurking there!
- Currently available hardware-based TTPs are prohibitively expensive.

In prior work, we have succeeded in using a limited-power hardware-based TTP for securing a particular two-party functionality on large data sets—Private Information Retrieval [15, 16]—by using careful algorithm design. This work led us to make two observations:

- Can we generalize this result, to make it easy to compute arbitrary multi-party functions in *tiny* TTPs?
- What if one designed a hardware TTP expressly optimized for these algorithmic techniques (thus improving efficiency) and discarded the notion of a rich general-purpose environment (thus reducing cost)?

[1] As is frequently the case, Denial of Service attacks are not prevented by the SCop security architecture.

This paper In this paper we report on our investigation of a hybrid approach to SFE, which uses a limited-power TTP to accelerate computations which fundamentally have a high overhead in the circuit model.

We started with the observation that in the traditional blinded circuit approach to general SFE, *indirectly-addressed* arrays are a major source of inefficiency. An array access like `A[i]`, where `i` is determined at runtime, has to be translated to $O(|\texttt{A}|)$ gates in the circuit (which essentially examine every element of `A`, and return `A[i]`). Thus a program with many indirect array accesses maps to a very large circuit.

However, our prior work with secure coprocessors adapted Goldreich and Ostrovsky's Oblivious RAM algorithm [13] to do secure array lookup efficiently (see Section 2.3). So, we introduced an *array gate* concept to capture these array lookups, and built a new compiler that turns programs expressed in Fairplay's Secure Function Definition Language (SFDL) into an arithmetic circuit augmented with array-access gates. We wrote a circuit evaluation program which runs inside a 4758 SCop, and evaluates a circuit (stored as a sequence of gates in topological order) generated by our compiler: it receives inputs from the two players (including array inputs), runs the circuit on them, and then sends the results to their respective recipients.

The SCop implements the array gates by using the private lookup algorithm from our previous work. In our design and prototype, we limit the SCop to only read a small fixed amount of data at a time (one gate and its inputs), in order to satisfy the security model presented in Section 3 and ensure our results will be applicable to smaller TTPs than the 4758.

To give an idea of how indirect array indexing works in the two models of blinded circuit evaluation and TTP-assisted circuit evaluation, we ran a simple SFDL program which sums an array of N 16-bit numbers. We report the performance of the two SFE implementations in Section 6.

We outline some future work in Section 8: speculation on the design of tiny TTPs optimized to support these operations, but which are otherwise minimal; and spreading our implementation over several TTPs so that none of them get a full view of the computation. As usual we close with a conclusion.

2 Background

In this section we describe the background techniques and tools which are important for the rest of the paper.

2.1 SFE via blinded circuits

Most current solutions for SFE center on scrambled or *blinded* evaluation of a combinatorial circuit representation of the function[2]. If Agnes and Boris wish to compute a function f on their separate inputs:

- They obtain a circuit C which computes f,

[2]Other representations have been explored too, eg. algebraic circuits and branching programs. See [22] for a summary.

- They assign one of them to blind C. Call the blinded circuit (which is not syntactically a circuit, it is quite different) BLIND(C). The other player will evaluate BLIND(C), without being able to learn anything about the actual values propagating through the circuit. Say that Agnes will blind the circuit and Boris will execute it.
- Agnes blinds C. In essence this involves replacing bit-values with random keys (called blinded bits), and replacing gates with tables which evaluate their corresponding gate but using blinded bits.
- Agnes sends BLIND(C) to Boris, together with the blinded bits corresponding to her inputs.
- Boris obtains from Agnes the blinded bits corresponding to *his* inputs using 1 out of 2 oblivious transfer (OT) [24, 6]. OT is necessary in order to hide from Agnes the value of Boris's input bits, while giving Boris (only) the blinded bit values corresponding to those bits.
- Boris evaluates BLIND(C), starting with the input blinded bits (he only possesses the correct ones) and using the blinded truth tables to obtain, in the end, the output blinded bits. During the evaluation he does not learn anything about the actual values he is propagating through the circuit.
- He sends Agnes's output blinded bits to her. She knows the correspondence of blinded bits to actual bits and can so interpret her result.
- Agnes sends Boris the blinding map for each of his output wires, so he can interpret his blinded results.[3]

Note that this protocol is not secure against a cheating Agnes. She could produce a blinded circuit which does not correspond to C, and in this way cause Boris to compute an incorrect result. Boris would not be able to detect such an attack. The Fairplay system currently addresses this weakness by having Agnes produce and send to Boris m versions of BLIND(C), using different secrets. Boris picks one to evaluate, and Agnes reveals the blinded wire values for all the others, so Boris can examine them. This allows Agnes a $1/m$ cheating probability, while increasing complexity by a factor of m. Many more sophisticated solutions to actively cheating players exist, eg. see [12].

2.2 Secure Coprocessors

A secure coprocessor is a small general purpose computer armored to be secure against physical attack, such that code running on it has some assurance of running unmolested and unobserved [34]. It also includes mechanisms, called *outbound authentication* (OA)[4] to prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [26]. The coprocessor is attached to a *host* computer. The SCOP is assumed to be trusted by clients (by virtue of all the above provisions), but the host is not trusted (not even its

[3] The last two steps can be done in reverse order too, so Boris learns his results first. Guaranteeing fairness is not easy [23].

[4] OA is more recently referred to as *attestation* in the context of the Trusted Computing Group.

root user). The strongest adversary against the schemes presented here is the superuser on the host, who may also be equipped with a drill.

IBM 4758 Secure Coprocessor

The 4758 is a commercially available device, validated to the highest level of software and physical security scrutiny then offered—FIPS 140-1 level 4 [30][5]. It has an Intel 486 processor at 99 MHz, 4MB of RAM and 4MB of FLASH memory. It also has cryptographic acceleration hardware, and notably a "fast path" DES and TDES mode of operation, where data can be streamed from the host through the device's DES engine without touching the internal RAM. The 4758 connects to its host via PCI.

In production, the 4758 runs IBM's CP/Q++ embedded OS; however, experimental research devices can run a version of Linux (as does the follow-on product from IBM, the PCIXCC [3]). Linux has considerable advantages over CP/Q++ in terms of code portability and ease of development.

The 4758 Outbound Authentication mechanism is based on a chain of keypairs and certificates, starting with the device keypair which is generated and signed at the factory. The mechanism allows software on the 4758 to prove its identity (program hash), the identity of all system software and the device attributes to remote parties. The direct software identity authentication could in theory be expanded along the lines developed recently for consumer-level trusted hardware like the Trusted Platform Module (TPM) [33]—eg. property-based attestation [25], and direct anonymous attestation [7], but this is orthogonal to our work here.

2.3 Hiding the array access pattern

An important security goal that the TTP must satisfy is to hide the access pattern of the program/circuit it is evaluating to the arrays that the program manages. We have experience with this problem, having worked on TTP-assisted solutions to the Private Information Retrieval (PIR) problem [15, 16]. PIR allows a client to retrieve item i from a database of N items without revealing i to the database operator—quite similar to the array access problem in this case. We make use of the classic Oblivious RAM *square-root algorithm*, introduced by Goldreich and Ostrovsky [13], to allow the TTP to access datasets on its host without revealing any information about the access pattern.

The basic idea of this algorithm is that the TTP randomly permutes the dataset D, such that it knows the permutation π but the host does not learn π, thus producing a permuted dataset D_π. Then the TTP can access elements in D_π directly, without thereby revealing the actual indices it is accessing.

The need to hide the relation between different accesses[6] requires the TTP to re-do every past access in addition to the current one, keeping just the needed value internally. Observing this, the host cannot learn which is the actual current access, and how it relates to actual past accesses.

[5] FIPS 140-1 has been superseded by 140-2 since 2002, but the new standard does not provide any higher assurance levels [31].

[6] In the absence of countermeasures, the host can learn whether two accesses refer to the same index or different indices, even if the actual indices are hidden by the permutation.

Clearly the complexity of this procedure grows with every access, so after some number r of accesses, the TTP produces a new permuted dataset to work with.

We permute arrays using Batcher's sorting network [5], which takes $O(N \log^2 N)$ time. This determines the optimum value of r (which minimizes the amortized overhead of the oblivious retrieval algorithm) to be $O(\sqrt{N} \log N)$. Thus, we re-permute arrays after $\sqrt{N} \log N$ accesses to them, and every access costs $O(\sqrt{N} \log N)$.

3 Security Model

We assume that the TTP itself is fully trusted by the players, but is also *tiny*, as we elaborate on in this section. (We use the 4758 for our prototype, since it is an existing SCop, but we limit our assumptions so the work will apply to future-generation smaller devices.) The TTP is associated with a *host* machine which provides all the larger storage and network communication for the TTP. The host is untrusted, the implications of which follow in Section 3.2. In this setting, we have to ensure that the security requirements of SFE are met: each player learns no more than what he can (feasibly) infer from his own output, and external parties learn nothing.

3.1 Tiny TTP

Since the TTP should be trustworthy in an environment where it is exposed to physical attack, it needs heavy armoring against such attacks. This makes it difficult to provide lots of RAM, never mind secondary storage, in the armored perimeter. The 4758 for example has 4MB of RAM, and even with just that, it costs about US$2500.

Our hypothesis is that the way to make trustworthy TTPs more accessible is to make their protected area as small as possible, and hence as cheap as possible. Accordingly we base our designs on the assumption that the protected space in the TTP is tiny. More concretely, we require the following amount of protected storage:

- $O(\log N)$, where N is the size of the largest sequence in the current program. This is to allow the TTP to manipulate a constant number of pointers into any of the datasets it is working on.
- $O(B)$, where B is the block size of the symmetric cipher used to encrypt sensitive data stored on the host. Larger data items are processed in pieces; for example if the tiny TTP needs to re-encrypt a data item which is larger than B under a new key, it does this block by block

We have described our concrete vision and design for a tiny TTP device more specialized and optimized than the 4758 [17], and plan to collaborate with computer architecture colleagues to prototype the designs. We also sketch this design in Section 8.

3.2 Untrusted host

The TTP's host provides storage for all the data which the TTP uses, and also the network link between the TTP and the outside world. Since the host is untrusted (ie. accessible to the

adversary), it should not be able to learn anything sensitive about the TTP's computations by observing the network traffic or bulk storage I/O from the TTP. A simple consequence of this is that the TTP has to encrypt sensitive it writes out to the host or sends to the network. It also has to attach a keyed MAC or other authentication to data it stores on the host, to prevent tampering and replay attacks.

More involved than confidentiality and authenticity is hiding the TTP's access pattern to host-stored datasets. In our application of TTPs to secure evaluation of circuits, the TTP must hide from the adversary the access pattern to any arrays it is managing. Thus, if the TTP encounters an array read gate for `A[i]`, it cannot just read the ith element of the dataset representing `A` on the host—it is obliged by the requirements of SFE to hide *all* information not computable from the results of the SFE [7]. In this case the TTP has to hide i from the host, and also i's relation to other array indices in the program, if such relations are not known from the SFE program code.

3.3 Our system's compliance

Informally, our system running in the security model outlined here meets the requirements of SFE as it:

- Always encrypts intermediate data in the computation of a circuit if the data leaves trusted areas (Agnes's machine for Agnes, Boris's machine for Boris, and the TTP for both).
- Completely hides the access pattern to all arrays in the computation, by using the algorithm outlined in Section 2.3.

It does not need to hide anything about the circuit, as that is known by both players (if they mind external parties learning the circuit, they should host the TTP on one of their machines), and it does not need to hide the access pattern to the gate values, as these are generated and read in an order determined by the circuit and not its inputs, and thus known to both players in advance.

4 A Case Study: Dijkstra's Algorithm with Heaps

Optimization problems in general use indirect addressing extensively, and thus would benefit from our approach if they are to be computed using SFE. Here we analyze a particular optimization problem—finding a shortest path between two vertices v_1 and v_2 in a graph:

- Boris has a directed graph $G = (V, E, A)$ with edge and vertex attributes A,
- Agnes has two vertices $v_1, v_2 \in V$, as well as a weighting function $w : A \to \mathbb{Z}$

Agnes should learn a shortest path from v_1 to v_2, using her weight function, or perhaps just the cost of such a path; Boris learns nothing.

It seems quite plausible that Boris wishes to keep his graph private (if it is a commercial asset for example), and Agnes wishes to keep her weighting function secret, as a trade secret for example.

[7] see also [35] for concrete examples of how the RAM access pattern of a program can reveal significant information, even if the RAM contents are all hidden

As elaborated in appendix Section A, Table 1 lists the different running times of shortest path algorithms with the two SFE models.

Bellman-Ford	Non-private	SFE circuit	SFE cct/TTP
$O(EV)$	$O(E \log V)$	$O(EV \log V)$	$O(E\sqrt{V} \log^2 V)$

Table 1: Running times for the Bellman-Ford algorithm, non-private or private, and Dijkstra's algorithm with heaps using various strategies: without privacy provisions; using only a blinded circuit; and using a circuit and TTP for faster indirect addressing.

5 Implementation

Our system architecture is outlined in Figure 1. We refer to a numbered sequence of items stored on the host as a *container*. When Agnes and Boris wish to compute a function f on their two respective inputs:

- One of them compiles a circuit version of f, which they both examine to ensure it is correct.
- They authenticate the TTP, using outbound authentication in the case of the 4758 (see Section 2.2). Thus they establish that they are using an untampered device with the correct software. Also they get an authenticated and confidential session to the TTP for the rest of the protocol.
- They send the circuit to the TTP, which could be at one of their sites, or at a separate location[8].
- The TTP stores the gates in container GATES on its host, with an HMAC for each gate.
- The TTP prepares a container VALUES to hold the values produced by the gates.
- The TTP also prepares a container ARRAY-A for each array A mentioned in the circuit.
- Agnes and Boris each send their inputs to the TTP, who writes them (encrypted) into VALUES in the case of scalars, and into the appropriate array container in the case of arrays.
- The TTP evaluates the gates one by one, by:
  - reading in the gate g, with number g_i, and as always checking its HMAC,
  - reading in its inputs from the VALUES container. The inputs are either scalars or array names,
  - Performing the gate operation, as described next, and

[8] If they want to keep their function secret from others, and the TTP is at an external location, the TTP would need to hide the circuit and the access pattern to it, similarly to how it hides accesses to the array container

– Writing the output value into slot g_i in VALUES.

- At the end of the circuit, the TTP sends the values of the output gates to their respective recipients, via the secure channel they established at the start of the protocol.

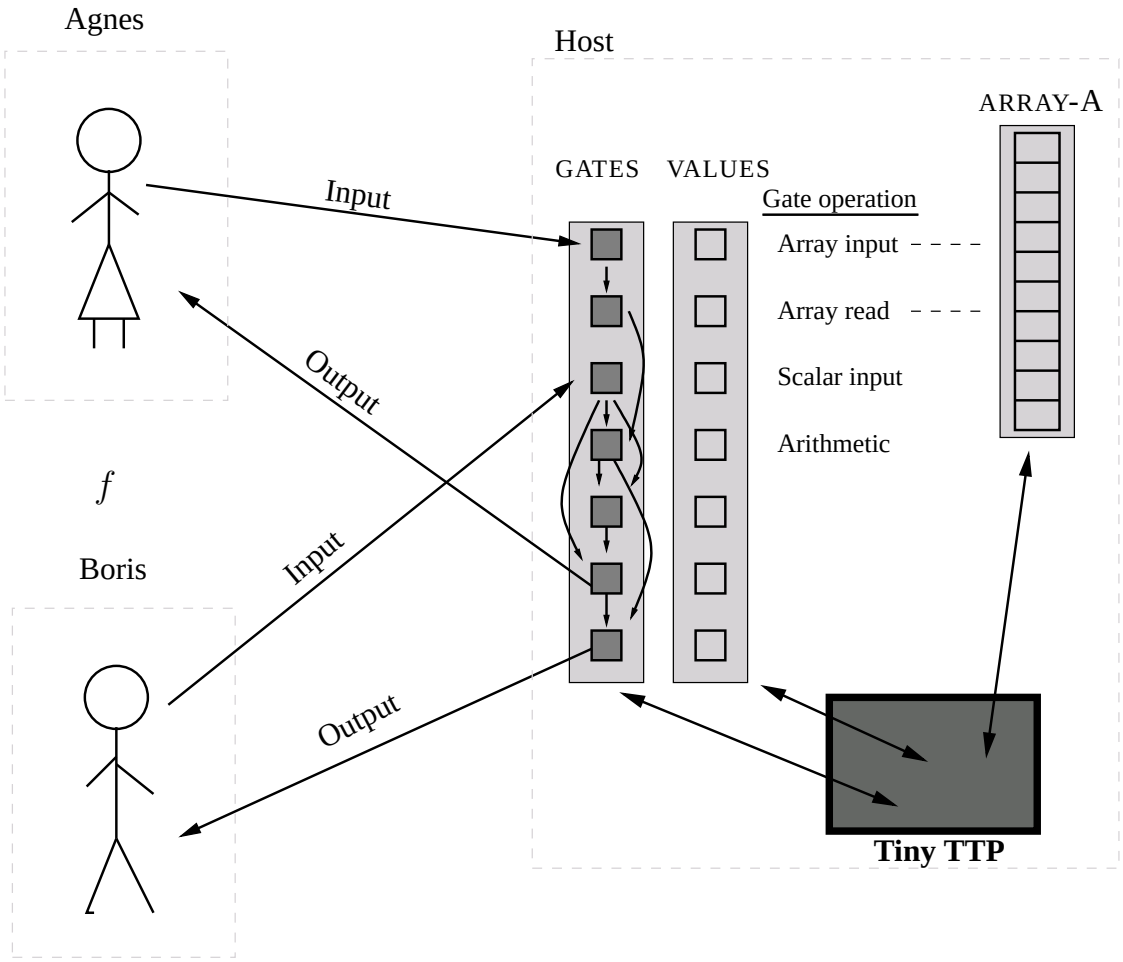

Figure 1: Our system architecture for TTP-assisted SFE. Agnes and Bob agree on a circuit representation of their function f (using a compiler). They send the circuit to the TTP. They also send their respective inputs to the TTP, who stores them (encrypted) as values of the input gates, or as values of an array container. Then the TTP computes the circuit gates in sequence, reading and writing gate values in the values container. The TTP executes array read and write gates using the square-root ORAM algorithm on the container representing the array (see Section 2.3).

5.1 Evaluating a gate

The TTP evaluates a gate according to its operation, and then writes the result, encrypted, into that gate's VALUES slot:

```
if (x != 0) {
    y = z/x;
}
else {
    y = z;
}
```

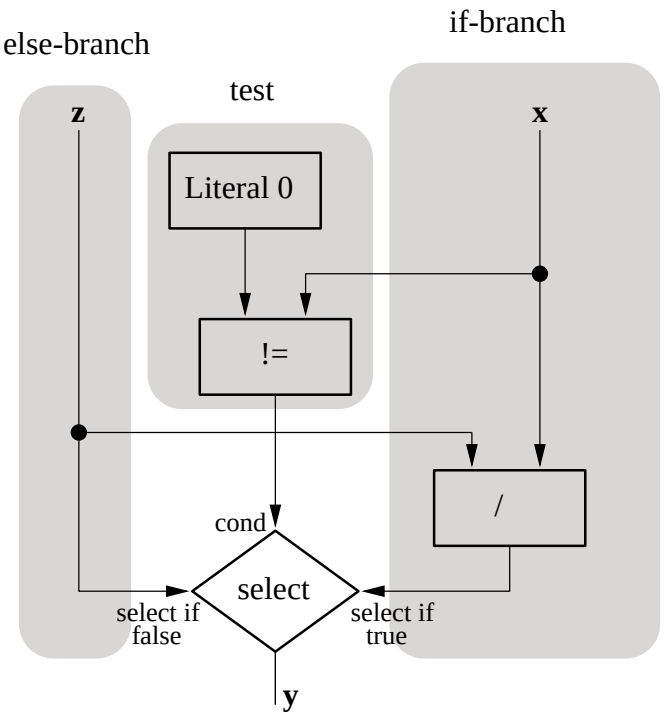

Figure 2: The generated circuit for the conditional code piece. Variables and other values are associated with wires, or equivalently gate outputs. The division is performed even if the divisor is zero, but in that case the value is discarded by the `select` gate.

Input The TTP has already entered the input value into the VALUES container before circuit evaluation,

Arithmetic Collect the one or two inputs from this gate's input locations, and evaluate the operation.

Array read Collect the inputs: array name and index, perform an ORAM retrieval (see Section 2.3) on the array container, and return the retrieved value.

Array write Collect the inputs: array name, index and new value; perform an ORAM update on the array container.

Select This gate selects one of two inputs based on a boolean value. It is used to implement conditionals in the SFDL source. The three inputs are: test value, value if test is true, and value if test is false. The TTP checks the test value and returns the selected value.

5.2 Implementing conditionals in the circuit

Evaluating a circuit involves evaluating every gate, whether its value is used in the end or not. In particular this means that both branches of a conditional expression are always evaluated. This means that the evaluator must expect illegal expressions, which need to be evaluated (and perhaps

yield a $\perp$ value). If the code is correct, $\perp$ values will be un-selected by a subsequent `select` gate. An example of a conditional division if the divisor is non-zero is shown in Figure 2.

Array writes inside a conditional also cause complications as they cannot be done on the original array container, in case the conditional selects the original and not the updated array. In this case we utilize a copy-on-write scheme to update a copy of the original array.

5.3 Compiler

We implemented the compiler from scratch in Haskell. We decided against using one of the two existing Fairplay compilers because they target a strictly boolean circuit (as required for the blinded circuit evaluation protocol), while we are additionally targeting arithmetic gates, and more importantly array access gates. In addition it seemed more sensible to start over seeing that this project should have several more stages, requiring further extensions to the compiler.

Like the current Fairplay compilers, it unrolls all loops in the SFDL code, and macro-expands all function calls. It does not do a single-assignment transformation but generates the circuit directly from the sequence of unrolled program statements. It trims the circuit to the gates which are reverse-reachable from the output gates, and outputs the remaining gates in topological order and numbered sequentially from zero. The gate format is intended to be as simple as possible for the C++ evaluator in the 4758 to read.

6 Results

In order to evaluate our hypothesis that the TTP-assisted circuit evaluation system is more efficient than a Yao-style blinded circuit evaluation, we ran an SFDL program which sums N 16-bit numbers in an indirectly-addressed array:

```
// 'ins' is an N-element array of pairs whose values are user inputs
for (i = 0 to N-1) {
    // sum the .y fields in order decided by the .x fields
    res = res + ins[ ins[i].y ].x;
}
```

One run of the program used the Fairplay version 2 compiler and evaluation engine, and the other used our compiler and evaluator in a 4758. The example is of course artificial, as one would normally sum the array in a fixed order, which Fairplay does efficiently, treating each array member as a separate variable. We show this example as a pure illustration of the difference in indirect array access speed, which can then be used to predict relative performance in other more realistic programs, like the Dijkstra shortest paths algorithm we discussed in Section 4.

The hardware setup for the Fairplay runs was: Alice and Bob[9] each on a separate machine, connected by gigabit ethernet, and each with a 2.7 GHz Xeon CPU, 4 GB memory, and SCSI drive. The results are shown in Table 2. They indicate that the TTP-based approach is indeed considerably more efficient than the blinded circuit approach, even with the latter on very fast machines.

N (array length)	FP gates	TTP gates	FP time	TTP time
64	193K	448	64	12
128	772K	896	255	26
256	3,085K	1792	1095	57
512	-	3584	-	142
1024	-	7168	-	372

Table 2: Circuit size and program execution time in seconds for different array lengths. The program simply summed the array of 16-bit integers, forcing the array addressing to be indirect, ie. it did N indirect accesses into an N-element array. The Fairplay implementation failed to evaluate its circuit for $N > 256$, due to insufficient memory (the JVM could use a maximum of 2.7 GB)[10]. Theoretically, we would expect the Fairplay running times to be $\mathrm{O}(N^2)$, and the TTP running times to be $\mathrm{O}(N\sqrt{N}\log N)$, which appears to be confirmed by these results.

To give an idea of how the two models of execution compare when there is no indirect-addressing involved, we ran a program which performs N 32-bit additions. The results are in Table 3, and indicate that the two execution models are quite similar. The blinded circuit evaluation system has an edge through much faster hardware, whereas the TTP-based evaluation benefits from larger (and hence fewer) gates and wires.

[9] Fairplay uses the more traditional character names

[10] It would be conceptually simple, though tedious, to make Fairplay evaluate its circuit without having the whole circuit in memory. Then the memory bound would go away, but that would be accompanied by an increase in running time due to all the extra disc activity.

7 Related Work

7.1 Trusted hardware

Current research (e.g., [19, 20, 32]) and product efforts (e.g., [14]) explore the notion of a secure computing environment built around the CPU alone. In contrast, the TCPA/TCG approach [33] attempts to secure an entire desktop, but not against physically present adversaries.

The XOM project [19, 20] investigated how to design a desktop-oriented processor architecture and operating system such that only the processor needs to be trusted, and not the OS and the RAM. The adversary's goal is to copy software which is run on the machine. They leave open the implications of the adversary observing a program's memory access pattern. In comparison, we are considering a very different use scenario—a specialized module, the tiny TTP, which does not have to provide for a full-blown multi-tasking OS, but which needs to hide *all* information about the software it is executing, and presumably be able to withstand a higher level of physical attack than a client-oriented system like XOM.

The AEGIS project [32] investigates the use of an innovative way for a processor to wield a secret key, by using a *Physical Random Function* based on random delays in silicon gates. It also assumes an untrusted RAM, but leaves open the consequences of exposing the RAM access pattern.

7.2 Other general approaches to SFE

The blinded-circuit solution to SFE is not the only one proposed. Other approaches make use of different representations of the function f to achieve different properties like limiting the communication burden on protocols with large inputs but sublinear communication in the non-private setting [22].

7.3 Specialized SFE protocols

A large body of work exists on designing specialized protocols for solving specific two-party or multi-party problems privately:

- Database operations, like `join` and `intersection`, across tables from two owners [2, 18];
- Privacy-preserving protocols for geometric problems [4, 11];
- Privacy-preserving data mining, eg. [10]

N (additions)	FP gates	TTP gates	FP time	TTP time
512	33K	513	17	9
1024	66K	1025	33	17

Table 3: Circuit size and program execution time in seconds for N 32-bit additions. Fairplay circuit adders for b-bit integers are constructed (as usual) with b full-adders each, where each full adder consists of 2 3-input gates. Thus the difference in gate count is about a factor of 64.

Although the majority of SFE work is theoretical, some of these works include concrete running time analysis [2] and even real prototypes [1].

8 Future work

8.1 Building Tiny TTPs

We are planning to work on a real prototype of a specialized device to optimize the kinds of algorithms we have used in this work and our earlier PIR work. The main differences from the current 4758 design will be:

- As explained before, the device will have very small memory and main processor,
- It will have a fast data channel to and from the outside,
- It will have a simple and fast processor in that data path, which can perform tasks like swapping data blocks based on some parameter in the blocks. This is to accelerate the execution of sorting networks and related oblivious networks.

Like the 4758, the new device will have a specialized symmetric cryptography engine in its fast path to and from the outside, to deal with the encrypting and re-encrypting needed for almost all data the device handles.

Our hypothesis is that the new design will increase the speed attainable currently with the 4758, while reducing the cost of the device, through minimizing the components inside the secure perimeter.

8.2 Reducing exposure to a single TTP

We will explore the potential to distribute the handling of an array among more than one TTP, such that no single one learns the values in the array, and (more challenging) the access pattern to the array. Also we are working on having the non-array parts of the computation be evaluated using circuit blinding, without the TTP, while the TTP handles the array accesses. Judging by the results in Table 3, there appears to be no speed advantage in the blinded circuit approach relative to the TTP, so the main motivation in keeping as much of the computation away from the TTP would be to reduce the players' exposure if the TTP misbehaves.

9 Conclusion

We have demonstrated that the incorporation of a trustworthy device into a two-party Secure Function Evaluation protocol brings about considerable efficiency gains. This is especially true for functionalities which are expensive in the circuit model, like indirect addressing of arrays. This efficiency comes at the cost of the protocol players having to trust a third party. The devices we considered are designed to warrant exactly this kind of trust, and through that they impose some challenges to their users, like small memory and slow processors, which we deal with. We believe that it is important that potential users of secure protocols have a range of choices in the

trade-off of performance vs. self-reliance, to meet their particular needs. Thus far the choices have largely erred on the side of self-reliance, and this work has attempted to provide a sample point in the performance corner. Investigation of this trade-off should continue.

Acknowledgements This work was supported in part by Cisco Corporation, the NSF (CAREER, CCR-0209144 and EIA-9802068), Internet2/AT&T, Sun, Intel, and by the Office for Domestic Preparedness, U.S. Dept. of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors.

Appendix

A Details on secure evaluation of Dijkstra's algorithm

Here we provide more detail on Section 4, and in particular the derivation of Table 1. In the discussion we focus on the computation of a shortest path, given existing edge weights. These weights would be computed (as part of the secure evaluation) using Agnes's w function before the shortest paths algorithm is run. The object representing w is likely to be large for any non-trivial function, meaning that it will also benefit from accelerated indirect addressing.

The Bellman-Ford algorithm solves the shortest path problem in $\mathrm{O}(VE)$ time[11], and is suitable for low-overhead circuit implementation as it accesses the adjacency list in a fixed manner, so no indirect addressing is required. Dijkstra's algorithm using a heap priority queue is considerably faster, taking $\mathrm{O}((E+V)\log V)$ time, which is $\mathrm{O}(E \log V)$ in the usual case of a connected graph [8], provided that indirect indexing into the heap is a constant-time operation. In general the time is $\mathrm{O}(E\, h(V) \log V)$ if indirect indexing into the heap takes $\mathrm{O}(h(V))$ time.

In the SFE scenario, if a direct circuit implementation is used for the heap, $h(V) = V$, so the running time degenerates to $\mathrm{O}(EV \log V)$. This is clearly worse than Bellman-Ford. If a TTP running a direct square-root ORAM algorithm is used for the heap, $h(V) = \sqrt{V} \log V$, yielding a total running time of $\mathrm{O}(E\sqrt{V} \log^2 V)$.

References

[1] Rakesh Agrawal, Dmitri Asonov, and Ramakrishnan Srikant. Enabling sovereign information sharing using Web Services. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 873–877. ACM Press, 2004. 14

[2] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 86–97. ACM Press, 2003. 13, 14

[3] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, May 2004. 5

[4] Mikhail J. Atallah and Wenliang Du. Secure multi-party computational geometry. In *WADS '01: Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 165–179, London, UK, 2001. Springer-Verlag. 13

[5] Kenneth Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Atlantic City, NJ, USA, April 1968. Thomson Book Company. 6

[6] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 547–557. Springer-Verlag New York, Inc., 1989. 4

[7] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM Press. 5

[11] As usual, we use V to indicate both the set of vertices and the size of that set, ditto for E.

[8] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein. *Introduction to Algorithms*. McGraw-Hill, second edition, 2001. 16

[9] J.G. Dyer, R. Perez, R. Sailer, Sean Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *Computer*, 34(10):57–66, Oct 2001. 2

[10] Alexandre Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. Privacy preserving mining of association rules. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 217–228, New York, NY, USA, 2002. ACM Press. 13

[11] Keith B. Frikken and Mikhail J. Atallah. Privacy preserving route planning. In *WPES '04: Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 8–15, New York, NY, USA, 2004. ACM Press. 13

[12] Oded Goldreich. *Foundations of Cryptography Volume 2: Basic Applications*, chapter 7: General Cryptographic Protocols, pages 599–764. Cambridge University Press, May 2004. 4

[13] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. 3, 5

[14] David Grawrock. LaGrande architecture. `http://www.intel.com/technology/security/downloads/scms18-LT_arch.htm`, September 2003. IDF Fall 2003 presentation. 13

[15] Alexander Iliev and Sean Smith. Private information storage with logarithmic-space secure hardware. In *I-NetSec 04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pages 201–216, Toulouse, France, August 2004. IFIP, Kluwer Academic Publishers. Workshop under 19th IFIP International Information Security Conference–SEC2004. 2, 5

[16] Alexander Iliev and Sean Smith. Protecting client privacy with trusted computing at the server: Two case studies. *IEEE Security and Privacy*, 3(2), March 2005. 2, 5

[17] Alexander Iliev and Sean Smith. Tiny trusted third parties. Technical Report TR2005-547, Dartmouth College, NH, USA, July 2005. to appear. 6

[18] Yaping Li, J. D. Tygar, and Joseph M. Hellerstein. Private matching. In D. Lee, S. Shieh, and J. D. Tygar, editors, *Computer Security in the 21st Century*. Springer, June 2005. To appear. 13

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000. 13

[20] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192, Bolton Landing, NY, USA, 2003. ACM Press. 13

[21] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In Matt Blaze, editor, *13th USENIX Security Symposium*, pages 287–302. USENIX, August 2004. 2

[22] Moni Naor and Kobbi Nissim. Communication preserving protocols for secure function evaluation. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 590–599, New York, NY, USA, 2001. ACM Press. 3, 13

[23] Benny Pinkas. Fair secure two-party computation. In *Advances in Cryptology—Eurocrypt '2003*, volume 2656 of *LNCS*, pages 87–105. Springer-Verlag, May 2003. 4

[24] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard, 1981. 4

[25] A. Sadeghi and C. Stuble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop*, September 2004. 5

[26] Sean Smith. Outbound authentication for programmable secure coprocessors. In *7th European Symposium on Research in Computer Science*, October 2002. 4

[27] Sean Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2005. 2

[28] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999. 2

[29] S.W. Smith, R. Perez, S.H. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*. National Institute of Standards and Technology, October 1999. 2

[30] National Institute Of Standards and Technology. Security requirements for cryptographic modules. `http://csrc.nist.gov/publications/fips/fips140-1/fips1401.pdf`, Jan 1994. FIPS PUB 140-1; URL current in June 2005. 5

[31] National Institute Of Standards and Technology. Security requirements for cryptographic modules. `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`, May 2001. FIPS PUB 140-2; URL current in June 2005. 5

[32] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, Madison, WI, USA, June 2005. 13

[33] Trusted Computing Group. `https://www.trustedcomputinggroup.org/home`, May 2005. 5, 13

[34] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994. 2, 4

[35] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 72–84, Boston, MA, USA, 2004. ACM Press. 7