Small, Stupid, and Scalable: Secure Computing with Faerieplay[*]

Alexander Iliev
Dartmouth College
alex.iliev@gmail.com

Sean W. Smith
Dartmouth College
sws@cs.dartmouth.edu

ABSTRACT

How can Agnes trust a computation C occurring at Boris's computer? In particular, how can Agnes can trust that C is occurring without Boris even being able to observe its internal state? One way is for Agnes to house C in a strong tamper-protected secure coprocessor at Boris's site. However, this approach is not scalable: neither in terms of computation—once C gets larger than the coprocessor, it becomes vulnerable to Boris again—nor in terms of cost. In this paper, we report on our *Faerieplay* project: rather than worrying about the limited size of a secure coprocessor, we try to make it as small as possible, with limited RAM and CPU. We start with the *Fairplay* work of Malkhi et al on implementing Yao's blinded-circuit solution to *secure multiparty computation* with software—this permits Agnes to trust C, but is too inefficient for all but small C. We then use our own prior work on using trusted third parties for practical *Private Information Retrieval* to design and prototype *tiny trusted third parties* (TTTPs) that substantially reduce the overhead involved in blind circuit evaluation.

Categories and Subject Descriptors

K.6.5 [**Computing Milieux**]: Security and Protection

General Terms

Algorithms, Experimentation, Security

1. INTRODUCTION

One definition of "trusted computing" is that Agnes can have assurances that Boris's machine is carrying out the computation correctly, without interference by Boris; a related one is that Agnes and perhaps other relying parties can verify that they are interacting with the genuine computation. These are the types of trust that the standard TCG approach based to TPMs aims to provide.

[*]This work supported in part via ISTS by US DHS, under grant 2006-CS-001-000001. Views are of the authors alone.

However, an even stronger property is that Agnes can trust that C is occurring without Boris even being able to observe its internal state. One way to provide this stronger property is for Agnes to house C in a strong tamper-protected *secure coprocessor* at Boris's site. For example, suppose Boris had an IBM4758 or 4764 installed and C was able to fit inside it (e.g., its data and code could reside entirely inside the device's internal storage). C could use the 4758's outbound authentication scheme [31] to create keypairs with certificate chains sufficient to attest to C's configuration and to establish authenticated communication channels to it—thus enabling Agnes to carry out her computation, with Boris learning nothing other than that a computation is taking place, its code, its duration, and the ciphertext of its communications (assuming, of course, that Agnes believes the platform's protections have not been subverted).

However, this approach is not scalable: neither in terms of cost nor computation. In terms of cost, devices such as the IBM4758 or IBM4764 are not cheap; the former retailed for $2K, and the latter for significantly more. However, in terms of computation, Agnes immediately runs into problems should C's code or data become larger than what fits inside the coprocessor. For example, the obvious solution of having the device page code and data out to Boris's host, acting as secondary storage, removes the opacity of the coprocessor: Boris can now see access patterns and other internal details of the computation.

In this paper, we report on our *Faerieplay* project: rather than worrying about the limited size of a secure coprocessor, we try to make it as small as possible, with limited RAM and CPU. We start with the *Fairplay* work of Malkhi et al. [26] on implementing Yao's blinded-circuit solution [36, 24] to *secure multiparty computation* with software—this permits Agnes to trust C, but is too inefficient for all but small C. We then use our own prior work on using trusted third parties for practical *Private Information Retrieval* to design and prototype *tiny trusted third parties* (TTTPs) that substantially reduce the overhead involved in blind circuit evaluation.

Section 2 reviews the basic building blocks of our approach. Section 3 presents the components we built, and Section 4 presents how they fit together. Section 5 presents our experimental evaluation, and Section 6 our theoretical. Section 7 discusses our more recent work prototyping custom hardware for the TTTP core. Section 8 discusses related work, and Section 9 concludes.

A preliminary snapshot of this project appeared as the work-in-progress paper [16] and related technical reports [21, 20]. A much longer treatment appeared as the first author's doctoral thesis [18].

2. BUILDING BLOCKS

2.1 Basics

We start by reviewing some tools from prior work that we use as building blocks.

Secure Coprocessors. As discussed briefly above, a *secure coprocessor (SCOP)* 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 [37]. It also includes mechanisms, called *outbound authentication* (OA) o prove that some given output came from a genuine instance of some given code running in an untampered coprocessor [31]. 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 root user). The strongest adversary against the schemes presented here is the superuser on the host, who may also be equipped with a drill.

Our initial experiments used the IBM4758, is a commercially available device, validated to the highest level of software and physical security scrutiny then offered—FIPS 140-1 level 4 [33]. It connects to its host via PCI. 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 device's internal RAM. The maximum throughput of the TDES engine is about 20MB/s, which is certainly much faster than contemporary (ie. about 1998) CPUs could manage.

In production, the 4758 runs IBM's CP/Q++ embedded OS; however, IBM also provides an experimental configuration with Linux (the follow-on product from IBM, the PCIXCC [3] runs Linux normally). Linux has considerable advantages over CP/Q++ in terms of code portability and ease of development, so we have used only the Linux configuration for our implementations.

The 4758 presents a general purpose computing environment, but it is geared towards providing high assurance for a *single* application, rather than supporting convenient multitasking. Thus, the software configuration at any one time consists of a single application. In terms of trust assurances provided by the 4758, the operating system and the user application can be considered as one: the 4758 assures that they will run as programmed, with secure access to their private key material. The device cannot assure that the application and OS are themselves correct—they could leak secrets or fail to preserve data integrity.

The 4758's casing includes a tamper-detecting mesh. If it detects an attempt to open the casing, it triggers zeroization of the DRAM (computation state), and battery-backed RAM (long-lived secret keys). Thus, a physically present adversary cannot open the device and learn anything useful about its computation. Additionally, the device will preemptively zeroize itself if it detects too low temperature or excessive radiation, both of which may prevent effective and fast zeroization of the device's RAM.

Secure Encryption. A symmetric encryption scheme $\mathcal{SE}$ operates on the space of keys K, plaintexts X and ciphertexts Y. We also explicitly specify the random bits R involved in the randomized algorithms. $\mathcal{SE}$ consists of three algorithms: $\mathcal{SE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ where $\mathcal{K}\colon R \to K$ is a randomized key-generation algorithm; $\mathcal{E}_K\colon X \times R \to Y$ is a randomized encryption algorithm using key K; and $\mathcal{D}_K\colon Y \to X$ is a deterministic decryption algorithm using key K. Informally, the security property of a secure encryption scheme is that, having seen a ciphertext, an adversary can only learn one thing about the source plaintext, which is its length. The formal statement of this security property allows the adversary to view encryptions of multiple plaintexts of his choice, and only requires that he has a negligible probability of inferring something (even one bit of information) about a "challenge" ciphertext. This security property is called *indistinguishability under chosen plaintext attack* (IND-CPA). Encrypting with an encryption scheme which provides IND-CPA is also referred to as *semantic encryption.*

Note that secure encryption does not only ensure that the adversary cannot reverse the encryption, it ensures that he cannot learn *anything* about the plaintext (apart from its length), including whether it is the same as the plaintext of some other ciphertext under the same key.

One immediate consequence of this security definition is that the encryption scheme cannot be deterministic, or the adversary could trivially learn if two ciphertexts correspond to the same plaintext. A typical way to achieve the required randomization is to use the CBC mode of encryption with a random IV [8, Chapter 4]; there are many other ways too, most involving different modes of operation with an underlying block encryption scheme like AES.

2.2 Oblivious RAM

Motivated by the problem of protecting software from copying and unauthorized use, Goldreich and Ostrovsky developed (theoretical) techniques for a trusted CPU to execute a program using an untrusted RAM, such that an adversary controlling the RAM cannot learn anything about the program [15].

The major challenge in foiling such an adversary is to hide the access pattern to the RAM, ie. the sequence of addresses issued to the RAM. This includes hiding the actual addresses, as well as relationships among them. More concretely, the access pattern that an adversary controlling the RAM observes should look the same to him, regardless of the program that the CPU is executing, and its inputs.

Two main algorithms emerge from this *oblivious RAM (ORAM)* work—the *square root algorithm* and the *polylog algorithm,* named by their amortized overhead. Let's define a *record* to be the amount of memory the CPU can hold inside at one time. In order to carry out one access to a RAM of N records, the square root algorithm takes $\mathrm{O}(\sqrt{N} \lg N)$ time, and the polylog algorithm takes $\mathrm{O}(\lg^4 N)$. Both times are amortized—the algorithms periodically engage in long pre-processing operations. Also, the big O is much bigger for the polylog algorithm—concrete operation counting shows it overtaking the square root algorithm only at about $N = 2^{20}$.

The square root algorithm works in two stages. First, the trusted party (the CPU, in the original ORAM model) pseudorandomly permutes the contents of memory. E.g., it selects a permutation π on the index range $\{1..N\}$ and relocates the contents of each record r, $1 \le r \le N$, to location $\pi(r)$, changing the encryption along the way. Using the terminology of Goldreich et al., the permutation algorithm must be *oblivious*: have the same I/O access pattern regardless of the input (ie. the permutation)[1]. [15]. In the second stage, the CPU makes $k \ll N$ retrievals. By now, the permuted dataset D_π is available to the adversary, and the CPU knows π. The CPU uses this knowledge to hide the identities of retrieved records. In order to retrieve record r, the CPU reads in $\pi(r)$ from D_π, and the adversary does not learn what r can be. What is left is to hide the relationships between retrieved items, so the adversary (for example) cannot tell how many times a given item was retrieved. The approach is to copy records which have been accessed into a *working pool* P_S of maximum size k, which is scanned in its entirety for every retrieval. On each retrieval for record r, one record from D_π is added to P_S: either r if it is not already there, or a random untouched record if it is. Thus, records in D_π are accessed at most once.

We do not discuss the polylog algorithm any further, as it was more expensive for the dataset sizes which are feasible to run on the TTTPs we were considering, and is also more complicated than the square root algorithm (the polylog algorithm can be considered a generalization of the square root algorithm).

2.3 Secure Multiparty Computation

Secure (multiparty) computation (SMC) aims to enable multiple parties to engage in a joint computation on their private or proprietary data. They would like to learn something which is a function of all their data, but none of them want to reveal their actual data to any of the others.

For example, a group could want to learn which of the members has the highest income, but no member wants to disclose their actual income to any of the others. This *millionaires' problem* is the "hello world" example of SMC.

SMC provides a starting point for the trusted computing model we aspire to: Agnes and Boris know C, each have input x_A, x_B (respectively), and each want to know $C(x_a, x_B)$ without the other knowing any details about the other one's input.

2.4 Blinded Circuits

In 1986 Yao devised the first protocol for two parties to securely compute a two-party function, such that each party only learns their own partial result [36]. Like most subsequent works in this area, his protocol works with the two-party function represented as a *boolean circuit*.

This protocol is attractive as it is reasonably simple, and is very efficient in terms of rounds—the participants only need one round of interaction, and the rest of the processing is done locally.

The basic idea of the protocol is that one of the parties, say Agnes, *blinds* or *scrambles* the circuit, to produce an object which can be evaluated similarly to a circuit, but using blinded values[2]. Agnes does not see any values besides her own input and output, and in particular she does not see any circuit internal values. Then the other party, Boris, evaluates the blinded circuit, during which he does know the blinded values being computed, but does not know what they mean, ie. what boolean values they correspond to.

Yao's protocol is formalized and proved secure against a semi-honest adversary in [24]. In describing the protocol, we start with Agnes and Boris agreeing on the function they will compute, as a boolean circuit C. They also agree that Agnes will blind the circuit, and Boris will evaluate the blinded circuit.

Circuit blinding. Agnes produces a blinded form of the circuit in this manner. First, for every wire w in C, she assigns a random secret k_w^0 to represent 0 and k_w^1 to represent 1. We call these *blinded bits*, keeping in mind that they differ for each wire w. They can in practice be random keys for a symmetric cipher like AES. Then for every two-input gate g in C, she builds a *blinded truth table* which will enable Boris to compute the output blinded bit given the two input blinded bits, without revealing any additional information. A simple way to do this is to represent each of the four entries in the truth table as double encryptions of the blinded output bit, using the two blinded inputs bits as keys. The four entries must be in random order, otherwise Boris could learn something about the actual values of the inputs for a gate, based on which table entry he ends up using to evaluate the gate. The resulting set of blinded truth tables constitute the blinded circuit BLIND(C); Agnes sends this to Boris.

Now, Boris has BLIND(C), but does not know any of the blinding keys. He also does not know any blinded values. In order to start the blinded computation, he needs the blinded values of the input gates. Getting Agnes's blinded inputs is easy: she knows the blinding mapping, so she can just blind her input bits and send Boris the blinded values. Getting his own inputs blinded is more complicated. Only Agnes knows the blinding mapping. In order for Boris to obtain from her the blinded bits corresponding to his own inputs, two security concerns must be addressed. Agnes should clearly not learn about the values of Boris's inputs, and Boris should not learn anything about the blinded values which do not correspond to his actual inputs. Eg., if for a particular input gate g his input is 0, then for gate g he should only learn the blinded value of 0. If he did learn other blinded values for the input gates, he could evaluate the blinded circuit with varying values for his inputs (recall that he now has blinded values for Agnes's inputs), and thus possibly infer something about Agnes's inputs.

These security requirements are exactly addressed by 1 out of 2 *oblivious transfer (OT)* [7]. Thus, Agnes and Boris engage in an OT protocol for each bit in the blinded values of Boris's input, with Agnes acting as sender and Boris as chooser. OT guarantees that Boris learns exactly one value—the blinded value for his input, and no more, which is what we need here.

Boris now evaluates the whole BLIND(C), starting with the input blinded bits and using the blinded truth tables to obtain, in the end, the output blinded bits. Boris 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. These two steps can be done in either order, and the order determines which player learns his or her result first.

[1]The access pattern, ie. the sequence and values of I/O operations, will not be identical for all π, but must look identical to a computationally bound observer.

[2]The terminology for the transformed circuit has varied, the most popular terms being "scrambled" and "garbled". We believe that "blinded" is appropriate as it captures the aspect of doing specific things with objects/strings whose actual value one does not know.

2.5 Blinded Circuits in Practice

The Fairplay project built a system with which two players can actually carry out Yao's protocol and compute a function securely between themselves [26].

Fairplay specifies a two-party functionality using a high-level imperative language called *Secure Function Definition Language* (SFDL). SFDL looks like a normal imperative language, and draws on the syntax of C and Pascal. The main components of Fairplay are a compiler to translate SFDL to a boolean circuit, and a runtime system which Agnes and Boris can use to carry out Yao's protocol: namely, blind the circuit, transfer input values using oblivious transfer, evaluate the blinded circuit, and distribute outputs.

The authors' experiments with Fairplay focused on evaluating SFE implementation techniques, like the OT protocol used and the techniques employed against active adversaries. The evaluation was in terms of computation as well as communication between the parties.

Unfortunately, self-reliant circuit-based solutions for SFE incur a communication and computation cost at least linear in the circuit size, and thus the circuit size is the primary performance metric. In addition, however, each gate is usually executed using an expensive algorithm or protocol in order to achieve strong security properties, thus increasing the cost even more.

Large constant overhead for scalar code. It is possible to compile some functions into a boolean circuit which can be executed serially[3] with only a constant factor slowdown compared to a RAM machine representation of the same function. For example, a scalar addition on 32-bit integers can be compiled to 64 gates (with 3 inputs each); a RAM program would require one 32-bit addition instruction. Likewise, any sequence of scalar operations has the same asymptotic cost in boolean circuit form as in RAM program form.

In these cases, the overhead of executing the function using a boolean circuit-based SMC protocol over executing the same function "normally", ie. on a RAM machine, is also a constant factor. Each gate (analogous to an instruction) works on single bits, and thus more gates are required than instructions which operate on words of 32 or 64 bits, Each 2-input boolean gate requires several cryptographic operations for execution. For example, in Yao's two-party protocol, each gate requires generation of two random symmetric keys; encryption of two ciphertexts under each of the 4 permutations of the keys; random re-ordering of the 4 ciphertext pairs; and decryption of (expected) 4 ciphertext blocks.

Linear overhead for array code. In addition to scalar operations, high-level code can use *indirect arrays.* An array access is indirect if the index is non-constant, and thus on a RAM machine the array lookup requires two lookups—one to get the index from RAM, and the other to read or update the array value at that index. An indirect array access is cheap on a RAM machine (disregarding issues of cache efficiency)—it takes constant time.

In the traditional blinded-circuit approach to general SFE, indirectly-addressed arrays are a major source of inefficiency. Each bit in the array is represented by a wire in the circuit, and the array lookup (or update) is translated to a $\lg N$ by N multiplexer. There is no more efficient way to encode the array and lookups against it in a one-pass circuit—if the array lookup sub-circuit included any fewer than N gates (of small constant fan-in), it would not be able to produce the value of some array index.

Thus, code which uses many indirect array accesses will translate to a very large boolean circuit, and take correspondingly long to evaluate. This is a serious problem, as indirect array access is frequently an essential component of efficient algorithms.

The existing SMC literature leaves open the problem of indirect array access, eg. [29].

2.6 Practical Private Information Retrieval

The problem of *private information retrieval* considers how a user can obtain a particular record from a large data set that a server offers, without the server learning anything about which record was requested Several ideas may come to mind about how to achieve PIR. One could be that if the records were somehow encrypted, the server could be kept in the dark about the retrieved records. To examine this further, note that for this scenario to make sense, the data records must have been generated by another party, who then provides them to the data server in encrypted form. However, this approach does not fully achieve the desired properties of PIR. In particular, it does not hide from server the relationships between retrievals: the server can always learn whether a request is for the same record as some previous request, and how popular records are.

Theoretical computer scientists developed many algorithms (e.g., [11, 10]) through which users, servers, and sometimes other parties could carry out computation and achieve PIR. However, these algorithms are not satisfactory in practice. Two threads of theoretical PIR work exist. The first assumes that the distribution server consists of k non-colluding servers, with replicated copies of the database. The other approach to theoretical PIR makes the more standard assumption of a single server which is computationally bound in the usual sense, eg. finds exponential algorithms infeasible. In this case, a best-of-breed scheme has poly-logarithmic communication ($\mathrm{O}(\log^4 N)$), but linear work for the server, with a large constant [10]. Concretely, the server's work is one mod-exp per bit in the dataset—too expensive for large N. For example, if $N = 10,000$ items of $8,000$ bits each, retrieving one item would need 80 million mod-exp operations, which take about 24 hours on a 2GHz AMD Opteron-280 64-bit server using openssl (`openssl speed rsa1024` reports 900 private key RSA operations per second).

Smith and Safford [32] then proposed the problem of *practical* PIR: using existing systems, can we provide PIR along the lines of a Web model: the user establishes an SSL session, issues a request, waits a short while, then receives the response? Their solution used commercial secure coprocessors and assumed that a coprocessor can only hold a fixed small number of records internally at one time. Asonov et al. [4] improved the Smith-Safford scheme by decreasing the processing time for a query at the expense of a periodic preprocessing step. We note that Asonov's algorithm is essentially the same as the "square root" algorithm developed earlier as one solution to the *Oblivious RAM* (ORAM) problem [15], with the SCOP playing the role of the CPU. In subsequent work, we reduced Asonov's preprocessing cost of $\mathrm{O}(N^2)$ to $\mathrm{O}(N \log N)$ by carrying out the permutation using switching networks [17], and then reduced the internal stor-

[3]meaning with a circuit simulator which executes the gates one by one

age requirements to O($\log N$) and added the ability to *store* records as well as retrieve them [19]. We also implemented these approaches, using an IBM4758.

The crux of our approaches here lie in using the TTTP to carry out the operations of a switch in a switching network, but in an oblivious way. The TTTP reads in two records (the inputs), switches their places if its switch-bit is set, and writes them out to the same two positions. The host should be unable to tell if the two records were switched or not. This can be achieved by re-encrypting the records for example, assuming semantically-secure encryption is used.

3. FAERIEPLAY COMPONENTS

Our Faerieplay system uses the Fairplay implementation of Yao's blinded circuit approach to secure multiparty computation as a starting point, but abstracts inefficient operations (such as array lookups) into new types of gates, moves these gate operations into TTTPs, and uses tricks such as our PPIRW scheme (Section 2.6) to implement them efficiently.

In this section, we present the system components we developed.

3.1 Languages

SFLD0. The first high-level language supported by Faerieplay was Fairplay's Secure Function Definition Language (SFDL). We wanted to be compatible with Fairplay so we could test execution of the same function on both systems.

As Faerieplay progressed, we found that diverging from SFDL in some ways was quite useful. From now we will refer to the Faerieplay SFDL simply as SFDL; we will refer to *Fairplay*'s SFDL as *SFDL0.* SFDL is strongly typed. Each expression and variable has a type. Variable types must be provided when the variable is declared–in this SFDL is similar to statically typed languages without type inference, like C and Java. SFDL semantics are similar to the C-family of imperative languages, with a sprinkling of Pascal: the result of a function is specified by assigning to a variable with the same name as the function.

SFDL. As we attacked more problems with Faerieplay, we found SFDL0 difficult to work with in some ways, and addressed the problems by adding these features to it. We added support for passing function parameters by reference, so that assignments to a parameter inside a function are visible once it returns. In the absence of this, the programmer has to resort to defining and returning structures containing all the modified variables. We enhanced `for` loops to count down as well as up, with the same syntax. The choice depends on which end-point is larger. We support a `print` statement which allows a snapshot of program state to be obtained, for debugging. We define outputs of the computation to be the return value of `sfdlmain`. SFDL0's entry point is the function `main`, and it passes inputs and outputs in specified fields in the structures which are passed to `main`. We do not specify a convention for naming parameters to the main function with the names of their owners, eg. Alice and Bob. Binding data owners to parameter names is left for a higher-level Faerieplay component.

(The syntax of our SFDL is fully specified in Appendix B of [18].)

FC++. We continued to increase the complexity and size of the problems we tackled with Faerieplay: e.g., electricity power scheduling auctions ([22, 28], and also 7.5 in [18]). At this point, our extensions to SFDL0 still left the development process quite difficult. We realized that we needed a different approach: enable reuse of an existing development toolset.

A way to do this in Faerieplay (and other systems with experimental execution environments) is to use a high-level language which can be passed both to our system (compiler, circuit machine, etc.) for specialized execution, *and* to an existing development and execution toolset for auxiliary tasks like debugging and code analysis.

We selected C++ for this job, and we call the resulting language *Faerie C++* (FC++). C++ has a rich enough syntax to provide all the functionality in Faerieplay. The C preprocessor allows us some flexibility in specifying our language, and patching up any difference to standard C++. Eg. FC++ *requires* a keyword `var` before a variable declaration. Such keywords make the grammar more regular and a custom parser's job much easier (in this case Faerieplay's parser). When compiling with a standard C++ compiler, we simply need a `#define var` somewhere in the program. C++ has a well-developed toolset. Finally, C++ is one of the most popular programming languages, so it is likely that a prospective Faerieplay programmer will already be familiar with it.

(The syntax of FC++ is fully specified in Appendix D of [18].)

3.2 Circuit and Circuit Virtual Machine

We present the specifications of the Faerieplay *circuit virtual machine* (CVM) and also the format of circuits given to the CVM to execute. We refer to the specification of the circuit as the *circuit format* (CF).

A circuit is conceptually a directed acyclic graph (DAG), with vertexes corresponding to gates of several kinds, and edges corresponding to data flow from gate outputs to subsequent gate inputs.

Our CVM supports several kinds of data on the circuit data wires (edges). First we will describe the different kinds of data types supported by the CVM, and then specify the bit format used to represent them.

Scalars. The most basic data types is an integer scalar, which is defined to be the same as a word on the underlying machine where the circuit virtual machine is run, eg. 32 bits on a 32-bit machine, or 64 bits on a 64 bit machine. In future work, we could decouple the word size specified by the CVM from the the word size of the native machine. This would increase implementation complexity, but improve portability of Faerieplay code.

Array references. Indirect arrays are represented in the CF as opaque references, with the same physical representation as scalars. The circuit machine can implement these references as it chooses, provided that it satisfies the required obliviousness properties. The CVM will have to maintain the actual array as well as any extra needed state (like a permuted version of the array) on the host, and will probably use the array reference as a pointer to the correct actual array to operate on. We'll refer to the state (on the host and in the TTTP) associated with an array as the *array representation.*

The CF specifies an important property of array references, which enables the circuit machine to implement array accesses efficiently: array references are *linear*, ie. they are used only once. Each array gate has as part of its result a new array reference, and a subsequent operation on that array must use the new reference.

This rule allows the CVM to perform array operations in place, without concern that a subsequent operation will expect the old array representation to be preserved. This license is significant, as an array read or write could trigger a full re-permutation of the array representation, and the CVM can safely perform this without worrying about preserving the previous state of the permuted array.

In contrast, there is no requirement that a scalar value be linear—a single gate's scalar value can be used by many subsequent gates, for example if multiple expressions refer to a single variable. This is safe as a gate's scalar value never changes after it is set.

Structures. The CF supports structures as first-class values, and provides a construct to extract fields from them—the SLICER gate, described below.

Array-read gates produce complete structure values. Fields are extracted by SLICER gates if needed by subsequent gates. Array write gates take a parameter which can limit the write to just part of a structure (usually one field); a whole structure value can also be written to an array.

NIL. What does the CVM do in the case of invalid gate inputs, like a zero-divisor, or an out-of-range array index? Since circuit evaluation should look the same outside the TTTP, independent of inputs; terminating execution is not an acceptable error-handling strategy, as it can tell the adversary when some particular event occurs, eg. when some value is zero. Thus, all invalid operations succeed, but return an error value, NIL. All gate operations are defined on normal values as well as on NIL.

Note that the security model requires that all values sent to the host (and thus visible to the adversary) be securely encrypted, and thus the adversary cannot tell if any given value is NILor not.

3.3 Gates

We outline the Faerieplay circuit gates.

Evaluating an (unclocked) hardware circuit involves evaluating every gate once, whether its value is used in the end or not. For example, a multiplexer could discard values produced by earlier gates, but those gates produce their values anyway. Faerieplay follows the same principle for most gates—all are evaluated, even if their result is subsequently discarded, in our case by a SELECTOR gate. For Faerieplay, this behavior is mandated by the requirement to make every execution look the same to the adversary, not because the CVM could not compute the circuit more efficiently. The CVM could just compute taken branches (and thus reduce its work), but this would provide information to the adversary.

Since both branches of a conditional expression are always evaluated, the evaluator must expect illegal expressions in the not-taken branches, like division by zero or out-of-range array access. Such gates are evaluated as always, but yield a NIL value. If the code is correct, NIL values will be unselected by a subsequent `select` gate.

We could use the same approach for arrays as for scalars: always carry out the reads and updates and keep all copies until the end of the conditional section, when we can discard the un-selected copies and continue with the selected ones. But doing this efficiently is complicated, and has runtime overhead (a factor of b with respect to the existing PIR/W overhead, where b is the branching depth—the number of active conditional branches).

Instead, our CF specifies a different treatment for arrays inside conditionals, which saves considerable work for the CVM. Array gates are supplied with an additional "gate enable" input, a boolean which specifies whether the gate is enabled or not, depending on the enclosing conditional values. Then during execution, only enabled array accesses modify the array, while disabled accesses perform a dummy operation. Using the PIR/W algorithm for array update, the CVM can easily make a dummy write look indistinguishable from a real write. The time cost of an enabled and disabled array access must of course be the same, or they would look different to the adversary.

Arrays are thus not selected on at the end of a conditional block, as the selection has been implicit during the execution of the branches.

We decided not to use the gate-enable approach for scalar gates, but stick to SELECT gates to implement conditionals. The main reason is that scalar SELECT gates impose no more overhead than maintaining an enable bit inside conditionals, and they provide a more modular approach to conditional evaluation than do gates with an enable input.

In Table 1 we describe the (slightly) formal semantics of the main gates comprising our circuit format: BIN, UN, LIT, READDYNARRAY, WRITEDYNARRAY, and SELECT.

Mostly the gates semantics are obvious, but the possibility of NIL values almost everywhere complicates the picture. The remaining gates (INPUT, SLICER, PRINT, INITDYNARRAY) have more of a support role, and thus do not require a formal definition.

3.4 Compiler

We decided to build our own compiler from scratch (rather than use Fairplay's). We anticipated the need to modify the compiler extensively for different purposes, which is easier to do with one's own code. Additionally, Fairplay's compiler had a hand-written recursive descent parser which seemed too inflexible on the front end. We did indeed modify our compiler extensively after its initial version, for several main purposes: We also wanted to extend Fairplay's language and then to support a second source language. The flexibility of a capable parsing tool was very helpful here, and allowed us to add the second language easily. Finally, we also wanted to produce different forms of output: the executable circuit, various circuit visualizations, and a circuit simulator which helped with debugging the CVM.

We implemented the compiler in Haskell, which is strong in manipulating tree-structured data, like abstract syntax trees. Haskell is additionally a very succinct and high-level language, which allowed us to enhance the compiler with fairly little effort. For circuit generation and manipulation, we extensively used the *Functional Graph Library* (fgl) which is included in the *Glasgow Haskell Compiler* (GHC) distribution.

We used the compiler structure laid out in Appel's *Modern Compiler Construction* series [2]. Guided by this framework, the implementation went fairly smoothly. The last step, circuit generation, is not covered in compiler textbooks.

The compiler makes several passes over the code: *parsing* using the BNF converter (BNFC), which defines Haskell types corresponding to the syntactic elements and automat-

Gate	Output		Comment
Bin[$\oplus$](x, y)	NIL	If$x =$ NIL or $y =$ NIL	Binary operator; Note that we propagate NIL generously; one could argue for (eg.) Bin[Eq](NIL, NIL) $\rightarrow$ true. Our current approach is defensive—less likely that two errors will produce a valid (non-NIL) result and this remain undetected.
	$x \oplus y$	otherwise	
	Except		
Bin[And](x, y)	true	If$x =$ true and $y =$ true	
	false	If$x =$ false or $y =$ false	This implements shortcutting And. Eg. consider `x != 0 && y/x > 1`. If x is 0, this should be false even though the RHS is NIL due to division by zero. Note that the parameters to `&&` could be reversed, and the same would apply.
	NIL	otherwise	ie. one true and one NIL.
Bin[Or](x, y)	true	If$x =$ true or $y =$ true	This implements shortcutting Or. Eg. consider `x == 0 || y/x > 1`. If x is 0, this should be true even though the RHS is NIL due to division by zero.
	false	If$x =$ false and $y =$ false	
	NIL	otherwise	ie. one false and one NIL.
Un[op](x)	NIL	If$x =$ NIL	Unary operator
	‘op‘x	otherwise	
Lit[i]	i		A literal scalar.
Select(sel, t, f)	t	If$sel =$ true	
	f	If$sel =$ false	
	NIL	If$sel =$ NIL	A NIL selector should only happen in an un-selected conditional branch, unless the program has an error.
ReadDynArray(A, i)	(A', NIL)	If$i =$ NIL	
	(A', NIL)	If$i \geq length[A]$	$A'[i] = A[i], \forall i$
	$(A', A[i])$	otherwise	
WriteDynArray[$slice$](A, i, x, e)	A'	If$e =$ false	e is an enable bit, set to false if gate is inside a non-selected branch.
	A'	If$i =$ NIL	$A'[i] = A[i], \forall i$
	A'	If$i \geq length[A]$	$\forall j, A''[j] \| slice = \begin{cases} x, & \text{if } j = i \\ A[j] \| slice, & \text{otherwise} \end{cases}$
	A''	otherwise	*slice* specifies which (contiguous) part of the array element to update. Used in case of updating part of a structure. **Notation:** $x \| slice$ is the part of x specified by *slice*, eg. one field from a structure.

Table 1: Gate semantics. A gate's static parameters are shown in square brackets (they can be considered part of the opcode), and the runtime parameters (which are obtained from earlier gates, with addresses specified by this gate's *inputs* parameter), are in parentheses.

ically produces an abstract syntax tree (AST) composed of those types; *type checking* the correctness of the AST, building symbol tables, and converting the AST to an *intermediate form*; *conversion to canonical form*, where some of the constructs in the source code are converted to a smaller number of canonical constructs; *function expansion and loop unrolling*; *circuit generation*; and *circuit cleanup* and *topological sorting.* (Chapter 10 in [18] provides more design and implementation details.)

3.5 Implementation and Debugging

Our initial implementation of the circuit virtual machine (CVM) consisted of a software application running inside a 4758. We settled on C++ as providing a good mix of runtime efficiency and high-level abstraction, which helps with achieving a correct implementation.

While developing example applications, we reached the situation where bugs in the actual SFDL program were more problematic than bugs in the compiler or circuit evaluator. Since the entire platform is experimental, there are no development tools like debuggers to help with this. Moreover, the circuit model of evaluation results in a very different, and rather less intuitive, order of execution than a RAM implementation. Thus, we had to provide ourselves with several compiler capabilities to help with development.

First we added a circuit simulator to the compiler, which would evaluate a circuit without any of the PIR/W complications. To help with the association between a particular gate and the corresponding SFDL source code, we introduced print gates. Debugging the circuit evaluator, and following traces of circuit execution, is helped along by having a clear visualization of the circuit to look at. We have the compiler generate a graph for two popular graph formats: uDraw(Graph) and Graphviz.

4. PUTTING IT TOGETHER

Faerieplay provides a secure multiparty computation service, whose users have to trust a *tiny TTP* to get similar security assurances as with traditional SMC protocols, but with much improved efficiency and scalability.

This is a summary of how a programmer, Peggy, would use both the Faerieplay and C++ toolsets to develop, debug and securely run an FC++ program.

First, Peggy writes FC++ code whose entry point is function `sfdlmain`. `sfdlmain` can have any parameters and return values. The Faerieplay compiler generates some boilerplate C++ code, to enable compiling and running the program with a C++ compiler and toolset. This helper code is just a `main` function customized for the program input and output structures. This generated `main` function interprets inputs to the program. These can come either as (string) parameters to `main`, or through a specified API which ac-

```
type Vertex  = struct { Idx num,
                        Idx edge_list_head,
                        Word num_out_edges,
                        Weight d,
                        Idx pi_idx,
                        Idx heap_idx
                        };

type Edge    = struct { Idx dest_idx,
                        Weight w };


type Graph   = struct { Vertex[V] Vs,
                        Edge[E] Es };
```

Figure 1: The graph representation in the SFDL implementation of Dijkstra. The types `Word`, `Idx` and `Weight` are all integers. In the `Graph` structure, The `Es` array is a flat array of edges, which holds all the edges in the graph. The adjacency list (ie. the outgoing edges) of vertex `V` is located contiguously within `Es`—it starts at `Es[V.edge_list_head]`, and occupies the next `V.num_out_edges` elements in `Es`.

cepts input from the user in some manner, eg. through the standard input. With the input data, `main` prepares parameters for `sfdlmain`, calls `sfdlmain`, collects its return value, and returns the output to the outside, eg. by printing it to standard output in a specified encoding.

Then Peggy compiles her program, together with the generated helper file, using a standard compiler like `g++`. She can debug this program using debuggers available in the C++ toolset. She can also perform other correctness verification, like information-flow analysis. At this point, she can be confident that her code is correct for the desired functionality. Finally, she compiles her program with the Faerieplay compiler to obtain a circuit, and runs it with the CVM to obtain the Faerieplay security properties.

5. EXPERIMENTAL EVALUATION

Most of our measurements and other experimental results are based on running a simple but complete implementation of Dijkstra's algorithm with heaps. We ran similar implementations on Faerieplay and on our Oblivious RAM prototype. Our SFDL implementation of Dijkstra's algorithm represents the graph as shown in Figure 1. The graphs we ran it on ranged in size from 7 to 255 vertexes. They were generated randomly with $5-10$ outgoing edges per vertex.

5.1 Against Oblivious RAM

To compare the Faerieplay circuit implementation against an implementation using oblivious RAM, we wrote a simple C implementation of the algorithm, and ran it on a MIPS emulator using an oblivious RAM.

We ran both implementations on graphs of various sizes. The graphs were randomly generated, with a fixed number of vertexes, and a random outgoing edge list from each vertex: the number of out-edges for each vertex was normally distributed around a fixed mean, between 5 and 8, and each edges's weight and destination vertex were uniformly selected. A random pair of vertexes was used as the source and destination inputs to the shortest path algorithm.

The hardware setup was the same in both cases—an IBM 4758 as TTP, running the TTP portion of the Faerieplay CVM. The host was a Dell Optiplex desktop, with a single Intel Pentium IV 1.8 GHz CPU.

	vertices $V=7$		vertices $V=15$		vertices $V=63$	
	Faerieplay	ORAM	***Faerieplay***	ORAM	***Faerieplay***	ORAM
time (mins)	**4.1**	174	**8.9**	300	**53**	1554
instructions/gates	**5.4K**	10.5K	**15.1K**	18.7K	**93K**	90K
mem/array reads	**640**	1.2K	**1.7K**	2.4K	**11.3K**	13K
mem/array writes	**510**	1K	**1.1K**	1.8K	**9.5K**	8K
RAM size (words)	**n/a**	4.8K	**n/a**	4.9K	**n/a**	6K

Table 2: Table of running times of Dijkstra on Faerieplay and ORAM

We recorded several measurements from the experiments. Note that many of the measurements for the ORAM implementation are not specific to Oblivious RAM but apply in general to the RAM machine form of the function. The measurements are shown in Table 2.

The measurements we obtained are for the most not surprising. ORAM has a larger RAM than Faerieplay has indirect arrays, and thus it spends much more time on oblivious access. The number of operations in each case is quite similar, which suggests that the Faerieplay circuit version is not burdened with overhead when compared to the optimized RAM executable (we compiled it with `gcc` at `-O2` optimization)

One surprising result is that both implementations generate a similar number of memory accesses. We would have expected that Faerieplay would have fewer indirect accesses than ORAM, because its indirect indexing is limited to arrays which require it. One explanation is that the circuit implementation performs some array accesses inside non-selected conditional branches. Those accesses are "dummies", but they still incur the cost of the indirect array access.

5.2 Against Fairplay

We could not run our Dijkstra SFDL code with Fairplay, as the Fairplay compiler does not support enough of the language we use (and, in our earlier experiments, Fairplay did not scale beyond small problem sizes). Instead, we used several indirect ways to compare the performance of the two systems.

We ran an SFDL program, defextreme-indirect-indexing, which sums N 16-bit numbers in an indirectly-addressed array, thus generating N indirect accesses to an N-element array. 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 our Dijkstra shortest paths example.

One run of the program used the Fairplay version 2 compiler and evaluation engine, and the other used the Faerieplay compiler and CVM in a 4758.

The hardware setup for the Fairplay runs was: Alice and Bob[4] each on a separate machine with a 2.7 GHz Xeon CPU,

[4]Fairplay uses the more traditional character names

N (array size)	gates		time	
	Fairplay	***Faerieplay***	Fairplay	***Faerieplay***
64	193K	**448**	64	**12**
128	772K	**896**	255	**26**
256	3085K	**1702**	1095	**57**
512	-	**3584**	-	**142**
1024	-	**7168**	-	**372**

Table 3: Faerieplay vs. Fairplay: circuit size and execution time for intensive indirect indexing. (Fairplay failed to evaluate its circuit for arrays larger than 256.)

N (additions)	gates		time	
	Fairplay	***Faerieplay***	Fairplay	***Faerieplay***
512	33K	**513**	17	**9**
1024	66K	**1025**	33	**17**

Table 4: Faerieplay vs. Fairplay: circuit size and execution time for scalar-only program

4 GB memory, and SCSI drive, connected by gigabit Ethernet.

The results are shown in Table 3. They indicate that the TTP-based approach is indeed vastly more efficient than the blinded circuit approach, even with the latter on very fast machines.

To give an idea of how Faerieplay and Fairplay compare when there is no indirect-addressing involved, we ran a program which performs N 32-bit additions. The results are in Table 4, 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.

6. THEORETICAL EVALUATION

We begin with the view of a single user of the Faerieplay system.

Agnes has a value $a \in A$. She also has a function $f : A \times B \rightarrow Y \times Z$. She wants to learn the partial value y of $f(a, b)$, where $b \in B$ comes from Boris. We temporarily hold off deeper analysis of "comes from Boris", beyond the intuitive understanding that b is Boris's input for the joint computation. Boris's view is symmetrical: he has a value b and the same function f, and wants to learn the partial value z of $f(a, b)$, where a comes from Agnes. Note that these are just the users' functional expectations, we will address their security requirements shortly.

In Faerieplay, Agnes will learn the result of the computation by sending a and f to the *Secure Computation Server* (SCS), which carries out the computation securely, and sends the results back. The main component of the server is a tiny TTP.

The TTP executes its program on its data by doing internal computation, and interacting with its host for bulk data storage. The host should be thought of as being under the adversary's control. During the computation, the adversary sees various intermediate data involved in the computation, any of which could potentially provide him with information he should not have. We will model this exchange of information with the adversary using a *transcript.* Note that a transcript is a standard tool used in security modeling.

DEFINITION 1. *The transcript should be a complete record of the data which the TTP exchanges with its host during a computation.*

We specify our transcript form aiming to categorize several kinds of data it contains, which will structure the subsequent security discussion. Thus, a Faerieplay transcript T consists of entries of the form $(\mathit{Time}, \mathit{Operation}, \mathit{Data})$.

At the end of a TTP-based computation of $(y, z) = f(a, b)$, Mallory has the complete transcript T for that computation. Note that this will usually include some entry corresponding to the results y and z, as they are communicated to the users via Mallory.

DEFINITION 2. *A TTP-based computation of function f on input (a, b) with resulting transcript T is secure against adversary Mallory if Mallory's knowledge about (a, b) is the same given T as it was before the computation, without knowing T.*

LEMMA 1. *A TTP-based computation of function f on input (a, b) with resulting transcript T is secure against a computationally bounded adversary A, if all entries in T are either (1) independent of (a, b), or (2) encrypted by the TTP with an IND-CPA secure encryption scheme. A can be passive or active.*

LEMMA 2. *The transcript generated by a Faerieplay computation is secure, according to the definition in Lemma 1. This applies both to a passive and active adversary.*

LEMMA 3. *Within the Faerieplay circuit and TTP-based model, an adversary can mount an active attack only by returning incorrect gate descriptions and gate values to the TTP.*

THEOREM 1. *Faerieplay's TTP-based circuit evaluation is secure according to Definition 2, against both passive and active adversaries.*

For full proof details, we refer the reader to Chapter 8 of [18].

7. FASTER AND SMALLER

As we noted, one of the motivations of this work was scalability: trying to preserve the "fully oblivious" nature of housing trusted computing inside a secure coprocessor when the computing is much larger than would fit inside such a hardware trusted third party. We approached this by giving up on putting significant computation inside the TTP, and instead tried to put as little as possible.

However, the commercial coprocessor we used for our prototype still has far more inside it that is strictly necessary and is rather expensive. Furthermore, it's not particularly good at the primary operation our TTTP must perform: quickly streaming data in and out that is both encrypted *and* integrity-protected.

In our preliminary work [21], we speculated on the potential of building special-purpose hardware for the core "encrypted switch" functionality necessary for our approach to practical private information retrieval and writing (Section 2.6), and speculated that a tiny TTP vastly smaller than a 4758 could vastly outperform it. In the subsequent software and theoretical work reported in this paper, our TTTP requirements generalized to arithmetic and other gates (Section 3.3) as well as the encrypted switches necessary for efficient array references.

More recently, we have been experimenting with prototyping such hardware (via an FPGA), and exploring additional optimizations such as pipelining the internal TTTP operations and running many TTTPs in parallel.

Due to space constraints (and disappearing authors), further discussion of these experiments will be left to a later paper.

8. RELATED WORK

Wang et al. [34] extended our TTP-assisted PIR/W work to improve the time, at the expense of more space inside the TTTP (and also provided a nice formal proof, which we used as model for ours). Williams and Sion have improved the polylog algorithm for oblivious RAM to produce a new randomized algorithm with much lower overhead [35]. Their projected performance of a PIR implementation using this algorithm on an IBM 4764 secure coprocessor.

The most relevant SMC implementation for this work is Fairplay (Section 2.5). A system to generate code for cryptographic protocols for functionalities like signatures and encryption is given in [25]. This is a much lower-level system than ours or Fairplay, but shares the aim of building a shared infrastructure for implementing secure protocols from an abstract description, and thus avoiding all the potential pitfalls of doing this by hand for every different scheme. A project aiming to produce a usable SMC system using the available self-reliant protocols is SIMAP[5]. They are developing a language for describing SMC functionalities [30], as well as a runtime system (called SMCR) to implement various self-reliant SMC protocols. The TrustedPals project aims to build an SMC system in a setting where each user has a smart card which is trusted by all the other users [13] for most purposes, and whose only failure mode is *general omission*, where they stop participating in the execution. Given this trusted network of security modules, they reduce the SMC problem to a fault-tolerance problem on the network of secure modules, and show solutions and impossibility results inherited from the fault-tolerance literature.

Remotely keyed encryption schemes seek to enable high-bandwidth encryption (on a host machine) using long-term keys held in low-bandwidth devices like smart cards [9]. This work shares the theme of enabling large computations using a small trusted space, but is otherwise quite different as it has no obliviousness requirements, and an adversary controlling the host can decrypt ciphertext until he is removed. On a similar theme, Modadugu et al. have developed a prototype using an untrusted host to help a Palm Pilot with the computation of generating RSA keys [27]. A large body of work also exists on designing specialized protocols for solving specific two-party or multi-party problems privately (e.g., [1, 23, 6, 14, 12, 5]).

[5]See `http://www.sikkerhed.alexandra.dk/uk/projects/simap.htm`

9. CONCLUSIONS AND FUTURE WORK

Beyond attestation and authentication, a stronger property of trusted computing is obliviousness: Boris knows no internal details of the computation taking place on his own computer, except for the fact that it's taking place and any inputs and outputs he is allowed. Providing this stronger form of trusted computing is straightforward with an armored secure coprocessor—but only as long as the computation fits inside it. In this paper, we presented our system for providing obliviousness for arbitrary large computations, via a compiler that transforms a high-level program to a circuit of elemental gates, and a circuit virtual machine that (executing inside a tiny trusted third party) executes this circuit obliviously. We demonstrated its practicality by executing Dijkstra's shortest path algorithm this way.

In (ongoing) future work, we plan to finish an FPGA prototype of a streamlined tiny trusted third party.

10. REFERENCES

[1] 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.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1997. Versions exist for C and Java as well.

[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.

[4] Dmitri Asonov and Johann-Christoph Freytag. Almost Optimal Private Information Retrieval. In *Privacy Enhancing Technoligies (PET2002)*, pages 209–223, 2002.

[5] Mikhail Atallah, Marina Blanton, Vinayak Deshpande, Keith Frikken, Jiangtao Li, and Leroy Schwarz. Secure Collaborative Planning, Forecasting, and Replenishment (SCPFR). In *Proceedings of 10th INFORMS Conference on Manufacturing and Service Operations Management (MSOM)*, Evanston, Il, USA, June 2005.

[6] 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.

[7] 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.

[8] Mihir Bellare and Phillip Rogaway. Introduction to Modern Cryptography. `http://www-cse.ucsd.edu/users/mihir/cse207/classnotes.html`, May 2004. URL live in Aug 2005.

[9] Matt Blaze, Joan Feigenbaum, and Moni Naor. A Formal Treatment of Remotely Keyed Encryption (Extended Abstract). In *EUROCRYPT '98*, volume 1403 of *LNCS*, pages 251–265, Espoo, Finland, May 1998. Springer-Verlag.

[10] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally Private Information Retrieval with

Polylogarithmic Communication. In *Eurocrypt 1999*, Prague, Czech Republic, 1999. Springer-Verlag. LNCS 1592.

[11] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–982, November 1998.

[12] 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] Milan Fort, Felix C. Freiling, Lucia Draque Penso, Zinaida Benenson, and Dogan Kesdogan. TrustedPals: Secure Multiparty Computation Implemented with Smart Cards. In *ESORICS 2006*, pages 34–48, Hamburg, Germany, September 2006. Springer. LNCS 4189.

[14] 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.

[15] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[16] A. Iliev and S.W. Smith. Faerieplay on Tiny Trusted Third Parties (Work in Progress). In *Second Workshop on Advances in Trusted Computing (WATC '06)*, November 2006.

[17] Alex Iliev and Sean Smith. Privacy-enhanced directory services. In *2nd Annual PKI Research Workshop*, pages 109–121, Gaithersburg, MD, April 2003. NIST.

[18] Alexander Iliev. *Hardware-Assisted Secure Computation*. PhD thesis, Dartmouth College, Hanover, NH, USA, 2009.

[19] 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.

[20] Alexander Iliev and Sean Smith. More Efficient Secure Function Evaluation Using Tiny Trusted Third Parties. Technical Report TR2005-551, Dartmouth College, Computer Science, Hanover, NH, USA, July 2005. `http://www.cs.dartmouth.edu/reports/abstracts/TR2005-551/`.

[21] Alexander Iliev and Sean Smith. Towards Tiny Trusted Third Parties. Technical Report TR2005-547, Dartmouth College, NH, USA, July 2005. `http://www.cs.dartmouth.edu/reports/abstracts/TR2005-547/`.

[22] Sherri A. Koenig. Power Generation Scheduling by Lagrangian Relaxation. Master's thesis, Cornell University, 1975.

[23] 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.

[24] Yehuda Lindell and Benny Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. *J.Cryptology*, 22:161–188, 2009.

[25] Philip MacKenzie, Alina Oprea, and Michael K. Reiter. Automatic Generation of Two-Party Computations. In *CCS '03: Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 210–219, New York, NY, USA, 2003. ACM Press.

[26] 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.

[27] N. Modadugu, D. Boneh, and M. Kim. Generating RSA Keys on the PalmPilot With the Help of an Untrusted Server. In *RSA Data Security Conference and Expo*, 2000.

[28] John A. Muckstadt and Sherri A. Koenig. An Application of Lagrangian Relaxation to Scheduling in Power-Generation Systems. *Operations Research*, 25(3):387–403, May 1977.

[29] 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.

[30] Janus Dam Nielsen and Michael I. Schwartzbach. A High-Level DSL for Secure Multiparty Computation. Linked at `http://www.brics.dk/~mis/papers.html`, 2006.

[31] Sean Smith. Outbound Authentication for Programmable Secure Coprocessors. In *7th European Symposium on Research in Computer Science*, October 2002.

[32] S.W. Smith and D. Safford. Practical Server Privacy Using Secure Coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001. (Special Issue on End-to-End Security).

[33] 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.

[34] S. Wang, X. Ding, R. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS 2006*, September 2006. LNCS 4189.

[35] Peter Williams and Radu Sion. Usable PIR. In *NDSS 2008*, February 2008.

[36] A. C. Yao. How to Generate and Exchange Secrets. In *FOCS 1986*, pages 162–167. IEEE, 1986.

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