CERIAS Tech Report 2005-58

EMPIRICAL EVALUATION OF SECURE TWO-PARTY COMPUTATION MODELS

by Marina Blanton

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Empirical Evaluation of Secure Two-Party Computation Models

Marina Blanton
Department of Computer Science
Purdue University
mbykova@.cs.purdue.edu

Abstract

Secure multi-party protocols make the computation of answers and decisions that depend on multiple parties' private data possible, without revealing anything about the private inputs (other than what unavoidably can be deduced from the outputs). There are general results showing that any probabilistic polynomial time function can be computed in this framework in an asymptotically efficient manner, using circuit-simulation techniques. There is a frequent belief that these general circuit-simulation techniques are not practical compared to custom-built (i.e., problem-specific) solutions, unless the function being computed has a naturally circuit-like formulation. This paper carries out a quantitative empirical evaluation of this belief, for a problem that would apparently benefit from a custom-built protocol (forecasting using time series techniques). Our findings are somewhat surprising in the following aspects. First, the custom-built solution does not overcome the general circuit-simulation solution on a local network until the problem size becomes quite large. Second, relaxing (even slightly) the requirement that, instead of "nothing", the protocols reveal "little" makes possible dramatic performance improvements over the solutions for the more strict requirement (whether they are custom-built or based on general circuit simulations). Third, other aspects (such as, e.g., system resources available) play a significant role in evaluation of a computational model. This paper describes the subtle implementation issues involved with this evaluation, presents its results, and talks about the lessons learned that should be valuable in future deployments of this kind of technology.

1 Introduction

Secure multiparty computation (SMC) permits two or more parties to securely and jointly conduct computations without the need to share any private information with other participants, even though the cooperatively computed answers depend on all of the participants' private data. SMC has received extensive treatment from the theoretical community, starting from the early results of Yao in 1982 [27]. Since then, a large number of work was dedicated to generalizing the results (e.g., [16, 7, 8]), making them robust under varying adversarial models (e.g., [5, 6, 9, 25]), improving efficiency of the protocols (e.g., [11, 12, 18]), etc.

Even if a small subset of these results were implemented and used in real-life applications, it would have the potential to significantly impact the way certain applications operate: it would allow computations that cannot be carried out today, and also result in more efficient and collaborative interactions without the fear that proprietary or personal information would be misused for unintended purposes. For example, SMC can lower costs and increase benefits of online interactions between businesses, by enabling them to collaborate without the drawbacks of having to reveal their private data to their counterparts (who might be competitors in other contexts). SMC protocols have been designed for bidding and auctions, database and data mining computations, cooperative benchmarking and forecasting, trust negotiations, credit checking, and many other domains. As Goldwasser [17] said in 1997, SMC is "an extremely powerful tool and rich theory whose real-life usage is at this time only beginning but will become in the future an integral part of our computing reality."

Unfortunately, secure multiparty protocols have not yet found widespread use, probably due to the complexity of their underlying models, but also the common belief that they introduce significant communication and computational overheads. To the best of our knowledge, until very recently there was no SMC implementation that permits evaluation of a wide range of functions. The Fairplay system [21] developed in 2004 provides a way to build a boolean circuit for a function and securely evaluate the circuit in the two-party setting. This important work also demonstrated the practicality of the general circuit-simulation approach for computing, in the SMC framework, functions that have a simple boolean circuit implementation. Yet there is still a common belief that, for complex enough problems (problems that do not have a naturally circuit-like formulation), these general circuit-simulation techniques are not practical compared to custom-built (i.e., problem-specific) solutions. This paper carries out a quantitative empirical evaluation of this belief, for a problem that would apparently benefit from a custom-built protocol: forecasting using time series techniques.

One of the reasons why we chose this application is that collaborative forecasting is an important application where the participating businesses benefit from using SMC technology: they make better forecasts of, e.g., future demand resulting in better business decisions, because they avail themselves of more accurate demand signals. In addition, the kinds of computations used in these forecasting models are simple enough that they are likely to be used in many other applications. That is, the core of the protocols is the secure computation of the division operation. While many operations (such as summation, multiplication, comparison) have received substantial amount of attention in the cryptographic community and efficient techniques for secure evaluation of such functions exist, division is not one of them: division results in non-trivial boolean circuits as well as it is not easy to implement using secure multi-party computation techniques over a large group (e.g., arithmetic circuits). Moreover, division necessitates handling of floating point arithmetics which is not usually considered in secure multiparty computations.

Our implementation of the problem-specific solution is based on protocols presented in [3] for business forecasting. Our findings, and other contributions of this paper, are summarized as follows:

- One of the custom-built solutions does not overcome the general circuit-simulation solution on a local network until the problem size becomes quite large. Intuitively, the complexity of the circuit is not damaging enough for small values of ℓ. This tells us that a tool for SMC-based collaborative forecasting, and more generally for applications where division is a substantial part of the computation, needs to use the general circuit-simulation techniques for smaller values of ℓ.

- Relaxing (even slightly) the requirement that, instead of "nothing", the protocols reveal "little" makes possible dramatic performance improvements over the solutions for the more strict requirement (whether they are custom-built or based on general circuit simulations).

- Running time of secure function evaluation by far is not the only metric that should be used for evaluation of SMC approaches. For instance, if running a boolean circuit is faster in certain cases but requires tremendous system resources for its compilation, parties who want to conduct secure computation might prefer a less efficient but also less demanding computational model.

- Another contribution of this paper is a description of the subtle implementation issues involved with this evaluation, a detailed quantitative presentation of its results, and conclusions based on the lessons learned that should be valuable in future deployments of this kind of technology.

This study gives results only for two-party computations, even though the underlying architecture of our custom-built system is general enough to permit execution of a protocol that involves any number of players and can handle any structure of the computation. Since we compare our results with performance of Fairplay, which works only for two-party circuit evaluation, two-party setting is sufficient for the purposes of this work.

The rest of this work is organized as follows: Section 2 provides an overview of related literature. Section 3 describes three possible approaches to conducting SMC: problem-specific solutions (which include our implementation), boolean circuits, and arithmetic circuits. In section 4, we describe in more detail computations performed using our problem-specific approach and boolean-circuit solution of Fairplay. The results of the experiments are given in section 5, and section 6 comments on the lessons learned.

2 Related Work

While there has been a large amount of prior theoretical work on secure multiparty computations that investigate various aspects of such computations, there are very few actual implementations in this area.

Malkhi et al. [21] gives is a full-fledged system, called Fairplay, that implements secure function evaluation using boolean circuits in the semi-honest adversarial model. The system consist of a high-level language that permits specification of a function to be evaluated, a compiler that transforms the program into a boolean circuit, and an execution module that evaluates the circuit. Since we use Fairplay in our experiments, more about the system is provided in later sections.

MacKenzie et al. [20] describes an implementation that uses threshold cryptography for function evaluation. This work includes a compiler that generates protocols for secure two-party computation using arithmetic circuits. Usage of threshold encryption allows these protocols to be resilient against active adversaries, but the computation is limited to functions of a specific form. While this work covers an important class of functions, this approach cannot be used to conduct computations that are focus of this work (i.e., division). For functions that can be expressed as arithmetic circuits, however, it results in efficient implementation.

Ajmani et al. [2] gives an implementation of a general-purpose computation system, called TEP, that relies on usage of a trusted third party. While it permits carrying out secure multiparty computations, the goal of secure multiparty protocols developed in the literature was to conduct such computations without dependence on a trusted third-party service.

Other implementations related to general secure computations such as [26, 29] have goals and use approaches substantially different from ours. That is, the approaches used include automatic generation and verification of security protocols and secure program partitioning techniques that do not accomplish the goals of this work.

3 Architectures

3.1 Boolean circuits

Fairplay [21] is a system that implements generic secure function evaluation (SFE) using garbled boolean circuit evaluation. Fairplay allows one to express a function in a high-level procedural definition language, called Secure Function Definition Language (SFDL), compile the program into a boolean circuit expressed in Secure Hardware Definition Language (SHDL), garble the circuit, and then evaluate it using Yao's method of [28].

SFDL is expressive enough to permit evaluation of a large variety of functions. That is, SFDL supports basic data types (e.g., integer, boolean) and data structures (e.g., enumerated types, structures, arrays), a number of operators (e.g., addition, subtraction, boolean bitwise operators, and comparison operators), conditional statements, fixed-size loops[1], and function calls. The resulting circuit, however, is boolean, which means that all multi-bit values must be transfered to single-bit values composing a (possibly large) number of gates. While evaluation of each gate in a garbled circuit is rather efficient (only input wires require execution of an oblivious transfer protocol that involves expensive public-key cryptography), evaluation and, more importantly, transmission of circuits consisting of many gates is resource-consuming. For example, multiplication and division were not implemented as language primitives because for two ℓ-bit numbers they result in circuits of $O(\ell^2)$ gates. While Fairplay is a perfect tool for secure evaluation of simple functions, it might not be suitable for certain types of computations even if an effort is put into using minimal resources.

The design of Fairplay prevents malicious behavior of both players who participate in the computation. While prevention of malicious behavior by Alice, who evaluates a circuit, is guaranteed through the use of cryptographic functions, Bob's misbehavior is prevented through a cut-and-choose technique (i.e., Bob constructs m circuits and opens $m-1$ of them to Alice to show that he did not cheat; probability of cheating is therefore $1/m$). This implies that if a faster but less resilient alternative computational model is available, the players might favor this approach.

More information about performance of Fairplay and computations we conduct using Fairplay can be found in subsequent sections.

3.2 Arithmetic circuits

Since the introduction of secure multi-party computation by Yao [27] and the initial plausibility results, a lot of effort has been put into enhancing these results, in particular efficiency of such computation. For instance, [4, 10, 11, 14, 15, 18, 19] among others treat issues of the communication and round complexity of secure multi-party protocols under various adversarial models. This literature achieves efficiency by evaluating arithmetic circuits over large fields instead of using boolean circuits. Unfortunately, arithmetic formulas are limited to the operations of addition and multiplication. While the reciprocal of a number can be computed using multiplication via, for instance, Newton method [1], this computation still cannot be expressed as an arithmetic formula and will require expensive conversions to boolean circuits. Therefore arithmetic circuits cannot be used to carry out the computations we are interested in and are not included in our further discussion.

Recent work by Cramer et al. [12] introduced secure multi-party computation over rings that permits efficient evaluation of more general functions than arithmetic formulas. Their techniques have a potential of performing computations that we carry in this work efficiently and under stronger security assumptions, but this is an area of future work.

3.3 Problem-specific approach

Our implementation is built in Java and has capabilities of running any secure multiparty protocol. The design is modular and permits substitution of individual building blocks at different levels of abstraction. The general architecture is depicted in Figure 1. In the figure, SMC player *implements* a protocol. A protocol *uses* a sub-protocol, which in turn *uses* a component. Then, for instance,

[1]Inability of SFDL to handle variable size loops is not a limitation of the system but is rather a requirement, because oblivious execution is not permitted in secure circuit evaluation.

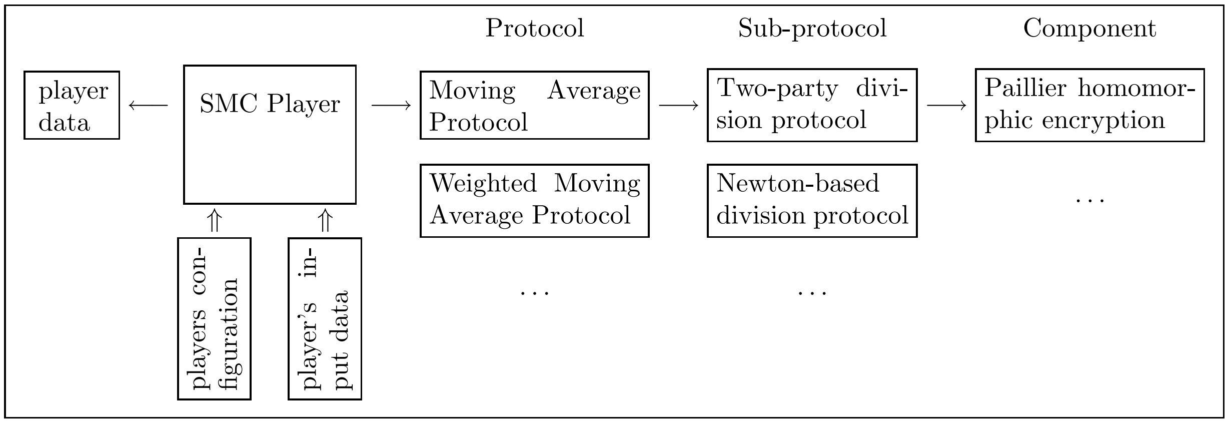

Protocol
Sub-protocol
Component
player data
SMC Player
Moving Average Protocol
Two-party division protocol
Paillier homomorphic encryption
Weighted Moving Average Protocol
Newton-based division protocol
players configuration
player's input data

Figure 1: Components of implementation of SMC protocols.

we could have an implementation of a moving average forecasting protocol using Newton-based division protocol, which itself uses Paillier homomorphic encryption scheme.

In order to determine configuration of the game, the *SMC player* takes *players configuration* as its input. This configuration contains information about the number of players and their locations. One of the players is assigned to be the *coordinating player*, and the software that executes the role of the coordinating player will ensure that all players are ready before the computation itself can begin. Each *SMC player* stores information about other players and their connections in *player data*. The underlying architecture allows us to execute a protocol of any structure and any number of players, but as a first step, we run only two-party protocols.

As was mentioned above, in this work we show performance of forecasting protocols, the core of which is the secure division protocol. Our implementation includes two forecasting protocols from [3], Moving average and Weighted moving average, each of which can use either of the division protocols available. The authors of [3] give a number of division protocols, all with their own advantages and disadvantages, and in this work we included two division protocols: a rather simple two-party division protocol and another two-party protocol that uses iterative process to compute the reciprocal of the divisor in order to perform division. These protocols assume *semi-honest* adversaries, i.e., those that will follow the computation as prescribed but might perform additional processing of the intermediate results in order to learn additional information. A more detailed explanation of the computations we perform is given in the next section.

Homomorphic encryption. Both of the division protocols rely on the use of a *homomorphic semantically secure encryption* scheme. A homomorphic encryption scheme is a public-key scheme that permits one to perform addition and multiplication operations directly on encrypted values. If we denote E to be the homomorphic encryption operation, such a scheme has the following properties: $E(x) \cdot E(y) = E(x + y)$ and $E(x)^y = E(x \cdot y)$. A homomorphic encryption scheme is *semantically secure* if $E(x)$ reveals no information about x, and thus $x = y$ does not imply $E(x) = E(y)$.

Currently, a number of such encryptions schemes are available, such as [24, 13, 23] and others. We use the original Paillier homomorphic encryption scheme [24] as our choice of homomorphic semantically secure encryption system. Efficiency improvements to the base Paillier scheme are available, but they only improve performance of the decryption operation. Since decryption is not used often in these protocols and such enhancements would not result in substantial improvements, we chose not to use them. Therefore, each encryption operation involves two modular exponentiations (one of which can be pre-computed) and each decryption operation involves one modular exponentiation.

Oblivious transfer. One of the division protocols implemented uses a sub-protocol that requires oblivious transfer (OT), and thus we implemented a recent efficient 1-out-of-2 OT protocol (OT^2_1) developed by Naor and Pinkas [22]. The protocol is implemented over a group $\mathbb{Z}_q$, which is a sub-group of of prime order q of $\mathbb{Z}^*_p$, where p is prime and $q|p-1$. Using their optimization techniques (where the same random exponent is used through different invocations of the protocol), this OT^2_1 protocol requires only one modular exponentiation per OT.

4 Computations

4.1 Custom protocols

As was mentioned before, our implementation consists of forecasting protocols. More precisely, we have implemented the moving average forecasting protocol, protocol 7 of [3], and the weighted moving average forecasting protocol, protocol 8 of [3]. In these protocols, pre-processing can be performed locally, and the only computation that requires interaction of the players is secure division protocol. In [3], the authors give a number of division protocols, and for the purposes of this work, we have implemented two division protocols: a fast secure two-party division protocol, protocol 5 of [3], and so-called secure Newton-based two-party division protocol, protocol 4 of [3]. For the sake of completeness of this paper, we briefly review these division protocols.

4.1.1 Division protocols

The division protocols given below compute x/y, where both x and y are additively split between the players, i.e., Alice has $x^{(1)}$ and $y^{(1)}$ and Bob has $x^{(2)}$ and $y^{(2)}$ such that $x = x^{(1)} + x^{(2)}$ and $y = y^{(1)} + y^{(2)}$.

Fast two-party division protocol. Protocol 5 of [3] is based on the use of homomorphic semantically secure encryption. It involves two rounds of communication and a small constant number of modular exponentiations, multiplications, and encryption and decryption operations. The main idea of the protocol is that one of the player blinds the inputs with random values, the other player performs the computation, and the first player then recovers the answer. A drawback of this protocol is that it may leak information when the dividend x of the protocol (which is split between the two players) is 0. Therefore this protocol might have a limited use.

Newton-based two-party division protocol. This algorithm first computes the reciprocal of y (i.e, $1/y$) and then multiplies the result by x. The reciprocal of y is computed using a technique derived from Newton's method. The algorithm starts with a rough initial approximation of $1/y$ and iterates to compute $1/y$ with the desired precision. More precisely, the iterative formula computes $z = \lfloor 2^{2\ell-1}/y \rfloor$, where ℓ is the (maximal) length of the numbers (x and y), using:

$$z_{i+1} = 2z_i - \lceil (z_i)^2 y / 2^{2\ell-1} \rceil$$

In this formula, z_i corresponds to the ith iteration of the algorithm, and the total number of iterations (and thus the number of communication rounds) is $O(\log \ell)$. To compute division by $2^{2\ell-1}$, this protocol executes a sub-protocol for secure scaling by a constant which involves one oblivious transfer[2]. Overall, this algorithm requires implementation of semantically secure homomorphic

[2]Note that simple shifting of the numbers will not result in correct computation because the numbers are split in modular arithmetics.

encryption and OT, and involves $O(\ell)$ communication rounds and $O(1)$ work per round.

4.1.2 Solutions and issues

In the rest of this subsection, we briefly describe solutions and issues directly related to the computations performed, as well as provide refinements to the protocols of [3]. Since, to the best of our knowledge, this implementation is the first of its kind, we comment on a couple of general implementation issues of SMC protocols, in addition to giving solutions specific to this work.

Estimating initial value of the reciprocal of an additively split value. In [3], the authors did not provide a method of estimating the initial value z_0 of the reciprocal of y. The difficulty in estimating this value is that y is split between the players but the initial estimate for $2^{2\ell-1}/y$ requires us to determine the value of the *first non-zero bits* of z. At the very least, we need to find the location of the first non-zero bit of z (if z_0 is determined incorrectly, the algorithm might take a larger number of iterations to converge to the right value or it may even not converge at all).

In this work, we use the following approach for determining z_0: using Fairplay we generate and run a small boolean circuit that computes

$$c = 2\ell - max(fnb(y^{(1)}), fnb(y^{(2)})) - 1$$

where $fnb(a)$ is the position of the first non-zero bit in the binary representation of a (counting from the least significant bit), and the length of the numbers the circuit operates is $\log \ell$. After computing c, z_0 is initialized to 2^c. This allowed us to compute correct results using only $\log \ell$ iterations of the algorithm, while the protocol given in [3] states a more general $O(\ell)$ number of rounds.

Notice that this circuit will give correct results only if both of $y^{(1)}$ and $y^{(2)}$ have the same sign. This is sufficient for our application because in all of the forecasting protocols the value of $y^{(i)}$ is guaranteed to be positive. In more general division protocols, however, a circuit that operates larger numbers (but which is still small and efficient) needs to be computed. Such a circuit will add $y^{(1)}$ and $y^{(2)}$ and determine the most significant non-zero bit of the resulting number.

Also note that the result of the circuit evaluation becomes known to the players. This information, however, can very likely be deduced from the result x/y, and this protocol assumes only passive adversaries (i.e., one of the players will not quit after learning the result of circuit evaluation).

Integration of protocols. As was mentioned earlier, the Newton-based division protocol uses a scaling by a constant factor protocol, which in turn uses OT as a sub-routine. Since all of these protocols work using modular arithmetics and have different requirements for their moduli, integration of such protocols becomes non-trivial. In our case, it is acceptable to operate additively split values with the same modulus T in both the division and scaling protocols. The oblivious transfer protocol, on the other hand, is used to communicate a pair of numbers in the range [0, $T-1$] and would require a modulus at least as large as T^2 to hide the numbers (i.e., 2048 bits long at the very minimum). If we chose the modulus of the OT in this manner, it, however, would result in unnecessarily expensive computations (as the size of the group in this algorithm is typically 160 bits long). Therefore, instead of directly hiding the numbers in the range [0, $T-1$] with OT, we keep the size of the OT modulus small and use it to communicate a secret key, which is then used to recover the pair of large numbers.

Negative numbers in additively split form using modular arithmetics. While negative numbers in a single protocol require careful treatment, the task of maintaining the sign of numbers

Integer length	8 bits	16 bits	24 bits	32 bits	48 bits	64 bits	128 bits
Number of gates	430	1622	3582	6310	14070	24902	didn't build

Table 1: Number of gates used for multiplication with varying integer sizes.

Integer length	8 bits	12 bits	16 bits	20 bits	24 bits	28 bits	32 bits
Number of gates	971	1379	3987	8377	12169	16665	didn't build

Table 2: Number of gates used for division with varying integer sizes.

becomes even more non-trivial when several sub-protocols and routines that use different moduli and non-linear operations are combined together. Careful treatment is especially important in case of division where dividing by a wrong number is generally unrecoverable. In our implementation we allocated the lower half of the range to positive numbers and the upper half to negative numbers and used adjustments to preserve the result of the computation.

Refinements to protocols of [3]. Since the details of these protocols are not crucial to the current discussion and description of the refinements requires additional background, they are provided in Appendix B.

4.2 Fairplay

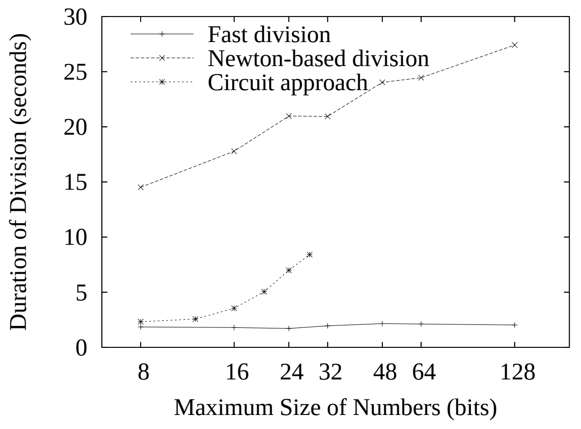
Similar to the previous case, computation of moving average forecasts will include local computation as the first step, and then execution of a division circuit to obtain the forecast. In order to obtain a better grasp on efficiency and gate complexity of boolean circuits generated by Fairplay, in addition to building division circuits, we built circuits for performing multiplication. Both division and multiplication program take input values in additively split form. Their 8-bit integer versions are given in Appendix A. These circuits are simple and do not handle negative numbers.

Tables 1 and 2 list the sizes of multiplication and division circuits, respectively, for varying lengths of operands. The multiplication program takes two operands x and y of the specified length ℓ in additively split form from Alice and Bob and returns their product $z = x \cdot y$ of length 2ℓ. The division program takes a dividend x and a divisor y of length ℓ in additively split form, and produces the quotient z of $\frac{2^\ell x}{y}$ of length 2ℓ — such a result permits computation of $\frac{x}{y}$ with the precision of ℓ bits after the decimal point if z is divided by 2^ℓ (and it has to be done outside Fairplay).

The main inconvenience in producing these programs was the unavailability of shift primitives in Secure Function Definition Language (SFDL) of Fairplay. We believe that the shift operation would simplify both multiplication and division programs and would likely result in reduced circuit sizes. Overall, implementing these functions as language primitives could produce more efficient circuits than those compiled from a SFDL program[3], because SFDL does not permit efficient low level operations (e.g., ability to access or assign a value to an individual bit of a multi-bit integer). Our effort to construct division circuits that manipulate integers at the bit level resulted in larger and less efficient circuits.

Compilation of these programs caused an interesting phenomenon: multiplication and division programs that resulted in circuits of approximately the same size had drastically different memory

[3]Multiplication was not implemented as the language primitive because of the cost of the operation. The authors instead recommend implementing it as a program.



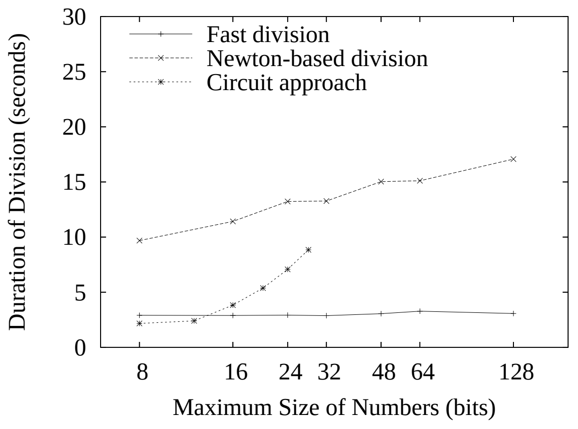


Figure 2: Performance of division algorithms on host 1 and host 2 using small input numbers with minimal network communication.

requirements for their compilation. That is, significantly larger resources were needed to build division circuits than multiplication circuits of the same size. For example, building a division circuit for 16-bit numbers required almost 1GB of RAM, while building a multiplication circuit for 24-bit or 32-bit numbers used only slightly over 100MB. The compilation process had up to 4GB of memory available for its use (i.e., the maximum amount a process can reference on a 32-bit architecture), and we still were unable to generate many division circuits. One might sacrifice precision of the result in order to build a circuit that divides at least 32-bit numbers.

Another unexpected finding was that Fairplay allows input integers to be no larger than standard 32-bit representation of integers, even if the input was specified in the SFDL program as having a larger size. We have also experienced obtaining incorrect results of computations in some cases, which brings an additional complication to the *current* comparison of empirical results of the two systems.

5 Experiments

This section shows the results of execution of three different approaches to secure division: custom fast division and Newton-based division protocols and division using a boolean circuit. Throughout the experiments, the custom protocols used the following configuration: the size of the group $\mathbb{Z}_n^*$ for Paillier encryption is an 1024-bit integer (or longer, to satisfy protocol requirements); the sizes of p and q parameters in the OT protocol were 1024 and 160 bits long, respectively; communication was performed using Java sockets with TCP connections. The authors of Fairplay [21] report the same settings for OT (using the same optimization technique) and also Java sockets with TCP. The experiments were performed using two machines: host 1 is a dual 2GHz Power Mac G5 with 2GB of RAM, and host 2 is a dual 1.8GHz Power Mac G5 with 1.5GB of RAM. The hosts reside on different networks with the average round-trip time (RTT) of 40–45 ms between them.

To minimize the impact of the network in our comparison, we first ran the experiments with minimal network overhead (i.e., on the same host). Figure 2 shows performance of the approaches executed on host 1 and 2, respectively, using the same input data of small length. Figure 3 shows performance under the same settings except that the size of the input integers grew to be close to the maximum size that the algorithms could operate. Each data point on all of the plots (now and on the consecutive plots) is the average over 10 executions of division only (i.e., excluding Java

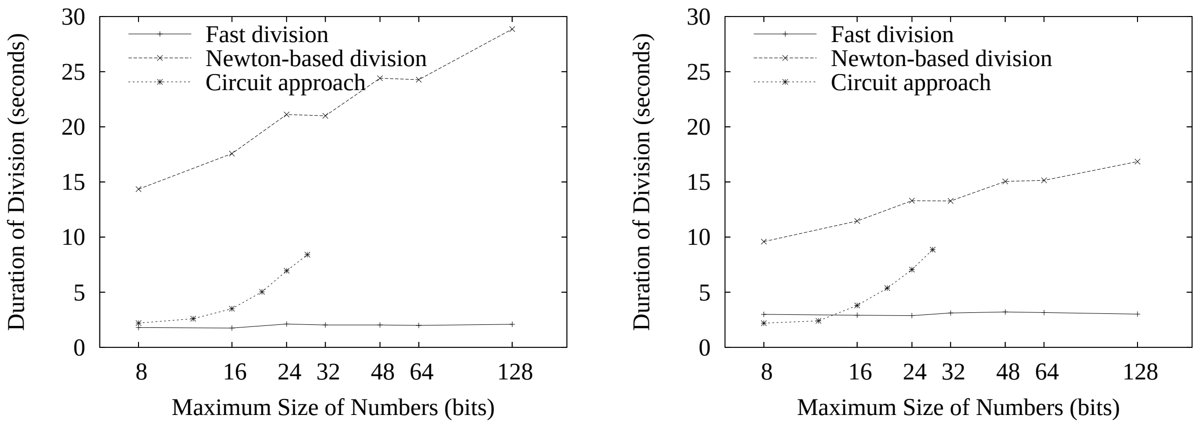


Figure 3: Performance of division algorithms on host 1 and host 2 using numbers length of which is close to the maximum length with minimal network communication.

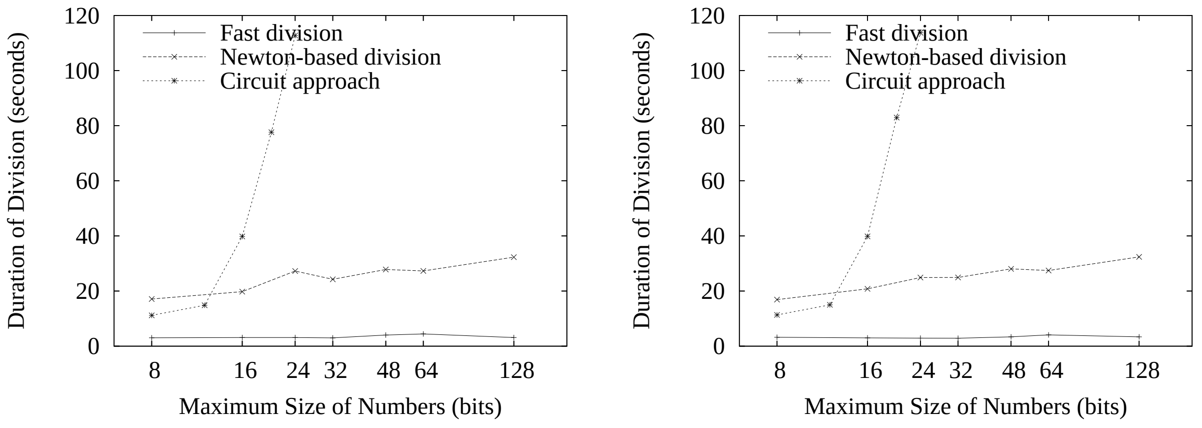


Figure 4: Performance of division algorithms with WAN communication using small input numbers and numbers length of which is close to the maximum length.

start up and program setup times). Also, the times recorded reflect the maximum of the times required to execute division at either party, which in cases of the custom division protocols was the second player (Bob) and in cases of the boolean circuits was the first player (Alice). As one can see, usage of large numbers does not introduce noticeable slowdown in the performance of the algorithms[4], and therefore there is no potential channel of information leakage.

Our explanation to the behavior of the functions is that in the custom division protocols the computation is dominated by the homomorphic encryption operations and the size of the numbers does not play a significant role. In the Newton-based division, however, the number of iterations of the algorithm depends on the size of the numbers (recall that it has $\log \ell$ rounds), and this is why the large number versions of the protocol result in heavier computations. A somewhat surprising result, however, is that in some cases increasing the size of the numbers does not result in slower

[4]Performance of the Newton-based division protocol appears to be slightly slower with large numbers than with small numbers at the 128-bit mark on host 1, but there is no other evidence of a slowdown, and therefore we do not treat this case as a pattern.

			8-bit	16-bit	24-bit	32-bit	48-bit	64-bit	128-bit
Host 1	Fast division	Alice	0.95	0.93	1.25	1.26	1.25	1.17	1.23
		Bob	1.79	1.75	2.11	2.03	2.03	1.99	2.09
	Newton-based	Alice	13.53	16.75	20.16	20.05	23.51	23.43	28.01
		Bob	14.35	17.57	21.12	21.00	24.41	24.28	28.87
	Circuit	Alice	2.20	3.50	6.94	N/A	N/A	N/A	N/A
		Bob	1.99	3.29	6.73	N/A	N/A	N/A	N/A
Host 2	Fast division	Alice	1.70	1.68	1.74	1.86	1.85	1.86	1.85
		Bob	2.99	2.91	2.87	3.11	3.21	3.15	3.01
	Newton-based	Alice	9.06	10.85	12.71	12.69	14.50	14.57	16.30
		Bob	9.59	11.46	13.30	13.28	15.05	15.15	16.85
	Circuit	Alice	2.19	3.80	7.05	N/A	N/A	N/A	N/A
		Bob	1.96	3.58	6.83	N/A	N/A	N/A	N/A
Mixed	Fast division	Alice	1.92	1.92	1.81	1.77	1.96	2.21	2.01
		Bob	3.22	3.00	2.93	2.88	3.39	4.09	3.37
	Newton-based	Alice	16.27	20.25	24.31	24.28	27.47	26.91	31.79
		Bob	16.90	20.81	24.91	24.94	28.04	27.45	32.38
	Circuit	Alice	11.34	39.84	113.74	N/A	N/A	N/A	N/A
		Bob	10.96	39.49	113.39	N/A	N/A	N/A	N/A

Table 3: Performance of the algorithms with large numbers using different configurations. In the mixed hosts example, Bob was running on host 1 and Alice on host 2.

computation. For instance, computation using 32-bit numbers is not slower than computation with 24-bit numbers (the same applies to 64- and 48-bit numbers, respectively). This brings another important point: in the Newton-based approach, precision of the result increases with the size of numbers. Therefore, even if the input data is always going to be very small, it might be advantageous to use a slightly larger maximum integer size than absolutely required because this will result in better precision of the answer. Using the experimental results, we can also see that lifting up the maximum size to a power of 2 comes free of cost.

Another notable observation is that execution times on host 1 and host 2 significantly differ, even though the machines have almost the same system resources. The fact that the fast division protocol is slower on host 2 than on host 1 but the Newton-based division protocol is, in turn, significantly faster must come from the operating system and/or Java settings (i.e., optimizations and such).

Results of the experiments with network overhead are shown in Figure 4. In these experiments, Bob was running on host 1 and Alice was running on host 2. As before, the plots correspond to the player who took the longer to execute. Table 3 lists a more detailed breakdown of the execution times for all of the experiments run with large numbers. As can be seen, network communication does not significantly slow down the custom-built protocols, but rather introduces larger randomness in the execution times. Secure function evaluation based on circuits, however, exhibits a drastically different behavior.

6 Lessons Learned

Even though in this work we considered only one particular application, our results allow us to make rather broad conclusions about secure two-party function evaluation and its different computational models.

First of all, this study showed that general results are not necessarily slower than problem-specific implementations. If there is no need to operate very large numbers, boolean circuits outperform one of our custom-built approaches on local area networks. Also considering a small possibility of leaking information, as in the second custom-built solution, allowed us to achieve fast results of the computation under all circumstances and all configurations.

Running time of secure function evaluation, however, by far is not the only determining factor in evaluation of alternative models. In fact, all execution times recorded during this study are small (with the longest being 2.5 minutes), and other factors play a more significant role in the large picture. What is important in considering what model to use (or finding that none of the existing models are suitable) based on our findings can be summarized as follows:

- *Availability of existing implementations.* If, for instance, no special-purpose solution is available and the player do not have the resources to create one, the general results will be used.
- *Adversarial model.* In our experiments, problem-specific solutions assumed a weaker adversarial model than circuit evaluation, and therefore these problem-specific protocols might be considered of limited use.
- *System resources* required to build a ready-to-use system. If, for example, it is impossible to build a circuit for division of 64-bit numbers, alternative solutions will be sought.
- *Network considerations* such as locations of players, available bandwidth, etc. There was no notable difference in performance of custom-built protocols with and without network overhead, but circuit evaluation was significantly slowed down by the network.

The fact that execution time is not a determining factor in choosing a computational model is also supported by our empirical data where apparently similar systems had rather different performance characteristics with Java custom-built applications (Figures 2 and 3). Experimental results are invaluable in determining behavior and characteristics of execution times as a function and, in general, feasibility of an approach.

This work is a first step in the direction of empirical evaluation of SMC computational models, and we hope that it will trigger more implementations and empirical studies of SMC that can initiate propagation of these valuable techniques into our every day life.

Acknowledgments

The author would like to thank Mikhail Atallah, Ethan Blanton, and Keith Frikken for valuable comments on this work. In addition, the author would like to acknowledge Jiangtao Li for implementating the oblivious transfer protocol and Joshua Blanton for providing the resources that allowed us to conduct WAN experiments.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] S. Ajmani, R. Morris, and B. Liskov. A trusted third-party computation service. Technical Report MIT-LCS-TR-847, MIT, May 2001.

[3] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara. Private collaborative forecasting and benchmarking. In *ACM Workshop on Privacy in the Electronic Society (WPES'04)*, pages 103–114, October 2004.

[4] D. Beaver. Minimal-latency secure function evaluation. In *Advances in Cryptology – EUROCRYPT'00*, volume 1807 of *LNCS*, pages 335–350, 2000.

[5] D. Beaver and S. Goldwasser. Multiparty computation with faulty majority. In *Symposium on Foundations of Computer Science (FOCS)*, pages 468–473, 1989.

[6] D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *Advances in Cryptology – Eurocrypt'92*, volume 658 of *LNCS*, pages 307–323, 1992.

[7] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, April 2000.

[8] R. Canetti. Universally composable securtiy: A new paradigm for cryptographic protocols. In *Symposium on Foundations of Computer Science (FOCS'01)*, pages 19–40, 2001.

[9] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure computation. In *Symposium on Theory of Computing (STOC)*, 1996.

[10] R. Cramer, I. Damgard, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *Advances in Cryptology – EUROCRYPT'00*, volume 1807 of *LNCS*, pages 316–334, 2000.

[11] R. Cramer, I. Damgard, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology – EUROCRYPT'01*, volume 2045 of *LNCS*, pages 280–300, 2001.

[12] R. Cramer, S. Fehr, Y. Ishai, and E. Kushilevitz. Efficient multi-party computation over rings. In *Advances in Cryptology – EUROCRYPT'03*, volume 2656 of *LNCS*, pages 596–613, 2003.

[13] I. Damgard and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, volume 1992 of *LNCS*, pages 119–136, 2001.

[14] M. Franklin and M. Yung. Communication complexity of secure computation. In *STOC'92*, pages 699–710, 1992.

[15] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computation with applications to threshold cryptography. In *PODC'98*, pages 101–111, 1998.

[16] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *ACM Symposium on Theory of Computing (STOC'87)*, pages 218–229, 1987.

[17] S. Goldwasser. Multi-party computations: Past and present. In *Invited paper to the Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC'97)*, pages 1–6, August 1997.

[18] M. Hirt, U. Maurer, and B. Przydatek. Efficient secure multi-party computation. In *ASIACRYPT'00*, volume 1976 of *LNCS*, pages 143–161, 2000.

[19] Y. Ishai and E. Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In *ICALP'02*, pages 244–256, 2002.

[20] P. MacKenzie, A. Oprea, and M. Reiter. Automatic generation of two-party computations. In *ACM Conference on Computer and Communications Secrity (CCS'03)*, October 2003.

[21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay – a secure two-party computation system. In *USENIX Security*, August 2004.

[22] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium on Discrete Algorithms (SODA'01)*, January 2001.

[23] T. Okamoto, S. Uchiyama, and E. Fujisaki. EPOC: Efficient probabilistic public-key encryption. Contribution to IEEE P1363a, 1998.

[24] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238, 1999.

[25] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Symposium on Theory of Computing (STOC)*, pages 73–85, 1989.

[26] D. Song, A. Perrig, and D. Phan. AGVI – automatic generation, verification, and implementation of security protocols. In *Conference on Computer Aided Verification (CAV)*, 2001.

[27] A. Yao. Protocols for secure computations. In *FOCS'82*, pages 160–164, November 1982.

[28] A. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[29] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.

Appendix A. Programs for Generating Boolean Circuits

A.1 Multiplication program

```
program Multiplication {
   const N = 8;
   type short = Int<N>;
   type long = Int<N+N>;

   type AliceInput = struct {short a, short b};
   type BobInput = struct {short a, short b};
   type AliceOutput = long;
   type BobOutput = long;
   type Output = struct {AliceOutput alice, BobOutput bob};
   type Input = struct {AliceInput alice, BobInput bob};

   function Output output(Input input) {
      var long result;
      var short a;
      var short mask;
      var long bShifted;
      var Int<8> i;

      a = input.alice.a + input.bob.a;
      bShifted = input.alice.b + input.bob.b;
      mask = 1;
      result = 0;

      for (i = 0 to N-1) {
         if (mask & a)
            result = result + bShifted;

         bShifted = bShifted + bShifted;
         mask = mask + mask;
      }

      output.alice = result;
      output.bob = result;
   }
}
```

A.2 Division program

```
program Division {
   const N = 8;
   const MAXIMUM = 65535;
   const THRESHOLD = 32768;
   type short = Int<N>;
   type long = Int<N+N>;

   type AliceInput = struct {short a, short b};
   type BobInput = struct {short a, short b};
   type AliceOutput = long;
   type BobOutput = long;
   type Output = struct {AliceOutput alice, BobOutput bob};
   type Input = struct {AliceInput alice, BobInput bob};

   function Output output(Input input) {
      var long[N+N-1] p;
      var long result;
      var long a;
      var short b;
      var Int<8> i;

      a = input.alice.a + input.bob.a;
      b = input.alice.b + input.bob.b;

      p[0] = b + b;
      a = a + a;

      for (i = 1 to N-1)
         p[i] = p[i-1] + p[i-1];

      for (i = N to N+N-2) {
         if (p[i-1] >= THRESHOLD)
            p[i] = MAXIMUM;
         else
            p[i] = p[i-1] + p[i-1];
         a = a + a;
      }

      result = 0;
      for (i = 2 to N+N) {
         if (a >= p[N+N-i]) {
           a = a - p[N+N-i];
            result = result + 1;
         }
         result = result + result;
      }
      if (a >= b)
        result = result + 1;

      output.alice = result;
      output.bob = result;
   }
}
```

Appendix B. Refinements to protocols of [3]

Here we briefly comment on the changes to the protocols of [3] our implementation caused us to make.

- In the fast two-party division protocol, four random numbers are selected such that they do not exceed the maximum integer value $MAXINT$. The range of the randoms, however, does not need to be bounded by this value: for added security, the upper limit should be on the order of $\sqrt{T}$, where T is the modulus in the encryption algorithm.

- Most of the division protocols of [3] divide two integer numbers. The weighted moving average protocol, however, uses floating point weights w_i, each of which is ≤ 1. These weights become a part of division computation and cannot be used unmodified. Therefore, the computation must be converted to integer arithmetic first (by, e.g., lifting the weights up and then dividing the result of the computation by the same value).

- Some optimization to the protocols that is not mentioned in [3] is possible. For instance, the multiplicative inverse of the divisor $2^{2\ell-1}$ modulo T in the Newton-based division protocol needs to be computed only once instead of on every invocation of the algorithm.