# Canon-MPC, A System for Casual Non-Interactive Secure Multi-Party Computation Using Native Client

Ayman Jarrous
University of Haifa
ayman@jarrous.net

Benny Pinkas*
Bar Ilan University
benny@pinkas.net

## ABSTRACT

This work intends to bring secure multi-party computation to the masses by designing and implementing a *browser-based* system that enables *non-interactive* secure computation. The system, denoted Canon-MPC for "CAsual NON-interactive secure Multi-Party Computation", is casual in the sense that participants do not need to install any software and do not need to agree on a time in which they all have to be online in order to run the computation. Rather, each participant can use a web browser to participate in the secure computation. The protocol is executed in a single pass between the participants. Each participant connects to a server once, without requiring other participants to be connected to the server at the same time. The system is appropriate for use by laypersons, since there is no need to install or configure any software except for a web browser.

The system is based on a protocol of Halevi et al. (Crypto 2011) for secure computation of symmetric binary functions, that is secure against malicious adversaries. We optimized the protocol using a batching technique for zero-knowledge proofs that greatly reduces their overhead.

We implemented a web site and client software for running the protocol, where the client was implemented using Native Client technology for running native code in a sandbox from within a web browser. We demonstrate that this technology is ideal for cryptographic applications. We describe experiments measuring the performance of the system. Lastly, we describe a variant of the protocol that can handle absentee parties, who were invited to participate in the protocol but did not show up.

## Categories and Subject Descriptors

D.4.6 [**Software**]: Operating systems—*Security and protection*

*Research supported by the SFEROT project funded by the European Research Council (ERC), and by an infrastructure grant of the Israeli Ministry of Science and Technology.


*WPES'13,* November 4, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2517840.2517845.

## Keywords

Secure multi-party computation, cryptography, native client

## 1. INTRODUCTION

Secure multi-party protocols were originally designed for a set of parties with symmetric capabilities, each with its own private input. The protocols enable the parties to compute any function of their inputs while hiding from each other everything but the final output of the function (and information that can deduced from it). The protocols require the parties to exchange multiple rounds of messages, where typically each party exchanges messages with all other parties.

The focus of our work is on enabling secure computation as a privacy tool for the masses. Namely, enabling casual users to easily use secure computation. In order to achieve this goal we had to overcome two major obstacles:

- Secure multi-party protocols require multiple rounds of interaction, and can be considerably slowed down if not all users are simultaneously online. Therefore, a protocol with a non-interactive communication pattern must be used.

- Casual users are typically uninterested in, or incapable of, installing and setting-up complex applications. We therefore implemented a web-based system where all interaction is done via a browser. This required implementing advanced cryptographic algorithms in a code that is downloaded and run in the browser.

The communication infrastructure of secure multi-party protocols, of multiple parties that are all exchanging messages with one another over multiple rounds, is different than the common communication pattern used on the web. Web communication is usually exchanged between clients and web servers, and exhibits a star-like communication pattern. Users often do not directly interact with each other. Rather, the common example is of users who communicate by posting messages on Facebook walls, or in bulletin boards.

Consider, for instance, a group of users who want to decide which of a few movies they should watch together, or want decide on a time for a meeting. If no privacy is needed then each user could post his or her preference on a wall or a bulletin board, and the users could count the votes after the last user has posted his or her message. If users wish to hide their preferences from each other but trust some other party, say Facebook or Doodle.com, with their inputs, then the personal preferences can be sent to this party, which

then tallies the votes and declares the final outcome.[1] If, on the other hand, the users wish to hide their preferences from each other as well as from any other third party, they could run a secure multi-party protocol for computing the preferred movie or the time for the meeting. A big obstacle towards implementing this computation is that most secure multi-party protocols require multiple communication rounds [11, 3], and ask each participant to send messages to all other participants, wait until responses are received from all participants, and then send additional messages (and repeat this for several rounds). This is in great contrast to the convenient non-interactive way in which users can use today non-private sites like Doodle.com.

*Interactive vs. non-interactive computation.*

There is a qualitative difference between non-interactive protocols, like the simple sharing of preferences when no privacy is needed, and interactive multi-party protocols. The former enable users to "send and forget" their inputs, while the latter require each user to wait until all other users send their messages, compute a message based on these messages, and repeat this process all over again multiple times. Each round ends only when the *last* user has replied, and therefore a single slow user can considerably affect the latency of the protocol. This is particularly true if some of the users are offline most of the time. (Assume, for instance, that one of the users is only online from 9 to 5, while another user is a night owl and is only available at night. Even a simple multi-round computation can then take days to complete.) It therefore seems that most multi-party protocols are incompatible with web based communication patterns. These MPC protocols assume that all parties are on-line throughout the computation and are able to communicate directly with each other. These assumptions are definitely not true for all potential participants of MPC protocols.

Our work is based the work of Halevi et al. [13], that initiated the study of secure non-interactive computation, provided appropriate (non-trivial) secure definitions, and designed corresponding protocols. These protocols included very efficient protocols for computing symmetric functions, and efficient but less practical protocols for computing any function. Protocols were presented for both the semi-honest and malicious settings. A review of this work, and of subsequent work (such as [12]) appears in Section 1.3.)

*Fully homomorphic encryption is insufficient.*

We note that even if the usage of fully homomorphic encryption ([9] and subsequent work) had been efficient in practice, it would not have immediately enabled non-interactive multi-party computation. Homomorphic encryption enables each party to encrypt its input, and enables the server to compute an encryption of the output of the function that needs to be computed. However, the private key that is required for the decryption of the output value must be shared between several parties, to prevent the server from decrypting the inputs of each party. Therefore, the decryption of the output value requires an additional round of communication involving the parties that share the decryption key.

[1]These sites do not currently support this private computation, where users' preferences are kept hidden from their peers. It will be rather easy to implement this functionality. Note, however, that this would not prevent the sites themselves from learning all inputs.

## 1.1 Our Results – MPC for the Masses

Our main goal in this project was to implement a system that enables casual non-interactive multi-party computation (Canon-MPC, for CAsual NON-interactive MPC). That is, the system should enable its users to run secure computation without complicated setup phases, and without downloading, compiling and setting-up any software. Any user of the system should be able to initiate a secure computation among his or her friends, or among any other subset of the registered users. In this respect this is the first MPC system that is suitable for wide usage by laypersons. Essentially, all that is needed in order to use the system is to access a web site and run a computation in the browser. It is also worthwhile to mention early on that the computation should provide a high standard of security, namely security against malicious (also known as active) adversaries.

It is instructive to compare the usage of Canon-MPC to that of previous systems for secure MPC. With former systems, such as, e.g., [2, 17], users have to download software and compile it. Then they have to setup the local copy of the software by entering the ip addresses of the other participants in the computation. Finally, all participants have to agree to be online at the same time in order to run the computation between themselves. In contrast, users of Canon-MPC have to register once at a web site. After registration, any user can initiate a secure computation by entering a function description and a list of user identifiers (essentially, email addresses). The web site notifies the invited users. Each of these users can then visit the site whenever it wishes, independently of the other users. After all users accessed the site, the output of the function is revealed to the users who participated in the computation.

A major observation that soon becomes clear is that an obstacle to the practical usage of the non-interactive mpc protocols of [13] is that a protocol that was initiated for a specific set of parties can only be completed if all these parties take part in the computation. Even a single party that fails to participate in the protocol prevents the other parties from recovering any output (even an output that is defined using some default input value for parties that did not show up for the computation). We address this issue in our work.

Our work contains the following contributions:

- **Canon-MPC**, a system for casual non-interactive secure multi-party computation. The system is browser based. Users only need to register once on a web site by providing an email address and a public key. Afterwards, each registered user can initiate a secure computation with any subset of the registered users.

  The system is available for usage at a web site, `canon-mpc.org`. The current implementation supports the computation of symmetric binary functions. That is, binary functions whose output does not depend on the identity of the those who provide inputs. Symmetric functions are particularly suitable for implementing decision making. The canon-mpc system is secure against malicious adversaries. The canon-mpc server, of course, does learn anything except for the output of the function, according to the security properties defined in [13]. The system provides an excellent performance in terms of the latency of the computation.

- **Batching zero-knowledge proofs.** The Canon-MPC system provides security against malicious adversaries (unlike many prior MPC implementations, such as FairplayMP [2], that are only secure against semi-honest adversaries). This level of security is based on the usage of zero-knowledge proofs. We improve the protocols presented in [13] with a batching technique that reduces the overhead of proving a set of $n$ Diffie-Hellman statements to be only weakly dependent of $n$. This greatly improves the overhead of the protocol, as is later observed in the performance measurements that we provide.

- **Using Native Client for MPC.** Secure multi-party computation requires each participant to run code locally. We found that Native Client technology is far superior for this purpose than installing applications, running Java applications, or using Javascript (see Section 1.2). Native Client enables building web applications which interact with downloaded code that is run inside the browser [20, 19, 1]. The downloaded code is a compiled C/C++ code, and as such can be very efficient. The user experience is very smooth, as the user only needs to access the canon-mpc web site and is not explicitly requested to download any code. We believe that Native Client is an ideal way for implementing secure computation, or other advanced cryptographic functionalities for web applications.

- **Handling absentee participants.** The system is typically initiated to run a computation between a certain set of parties. However, not all of these parties might actually contact the web server in order to participate in the computation. The basic protocols of [13] do not handle this case, since they require data from all the participants in order to decrypt the result of the computation. Otherwise the computation is left in limbo and cannot be completed. We address these issue by presenting a modified protocol that handles the case of absentee participants by running a limited second round of the computation. We also present arguments explaining why non-interactive computation is impossible in this case.

## 1.2 Tools

Our work is based on two major tools, the one-pass MPC protocols of [13], and Native Client technology.

### 1.2.1 Computing without simultaneous interaction

Halevi et al. [13] initiated the study of secure computation in a client-server model, where each client has only a single interaction with the server, independently of other clients. The basic setting consists of a server and $n$ parties, denoted $P_1, \ldots, P_n$, with respective inputs $x_1, \ldots, x_n$. The parties wish to jointly compute a function $f(x_1, \ldots, x_n)$. The server learns the output value. It is assumed that no two parties connect to the server at the same time, but the order in which the parties connect to the server can be arbitrary and does not have to be decided in advance. To simplify the notation we assume that $P_1, \ldots, P_n$ connect to the server in this order.

It was observed in [13] that a protocol in this model cannot provide the standard notion of security, even in the semi-honest model. This follows since a collusion of the last $n - i$ parties with the server is able to compute the residual function $f(x_1, \ldots, x_i, z_{i+1}, \ldots, z_n)$, where $x_1, \ldots, x_i$ are the input values that were already set by the first $i$ parties, and $z_{i+1}, \ldots, z_n$ are variables that can be set multiple times by the colluding parties and take any value. In other words, after the first $i$ parties complete their participation in the protocol, a collusion of the last $n - i$ parties and the server can simulate the continuation of the protocol any number of times, where in each time they can provide different inputs $z_{i+1}, \ldots, z_n$ and observe the output. This feature of one-pass protocols is inevitable, since the protocol must allow the last $n - i$ and the server to complete the computation of the function without any intervention of the first $i$ parties, and therefore a collusion of these $n - i$ parties and the server cannot be detected.

#### *One-pass decompositions.*

Security is formalized in [13] in the following way. First, a one-pass decomposition of $f$ is defined to be a vector of functions $\{f_i(y_{i-1}, x_i) \mid i = 1, \ldots, n\}$, where $y_{i-1}$ is the intermediate value of the computation after taking the inputs of the first $i - 1$ parties. A *minimum-disclosure* decomposition is one where the intermediate value $y_i$ contains no more information than the truth-table of the residual function that is defined by the inputs of the first $i$ parties. (That information, the truth-table, is inherently leaked to a collusion of the last $n - i$ parties, since they can compute the residual function as many times as they want. A decomposition is minimum-disclosure if it leaks no additional data.)

As an example, consider the sum function, which has the decomposition $y_0 = 0$, $f_i(y_{i-1}, x_i) = y_{i-1} + x_i$. In this case $y_i = \sum_{j=1}^{i} x_j$. This decomposition is minimum-disclosure since $y_i$ can be computed by setting the inputs of parties $P_{i+1}, \ldots, P_n$ to be all 0.

#### *Binary symmetric functions.*

Another example is the case of $n$-input binary symmetric functions, where the input contains $n$ bits, and the output depends only on the number of 1's in the input. Namely, the output of the function does not depend on the identity of the parties who have the different inputs. This family of functions includes many important functions, such as the AND, OR, PARITY, and MAJORITY functions. The truth table of a binary symmetric function is efficiently represented as a table of length $n + 1$, that for $j = 0, \ldots, n$ specifies the output of $f$ on inputs with Hamming-weight $j$. A one-pass decomposition of a symmetric binary function can be defined by setting $y_i$ to be the truth table of the function that is defined after the first $i$ parties have provided their inputs. The truth table is of length $n + 1 - i$. This decomposition is trivially minimum-disclosure.

An important property of this decomposition of binary symmetric functions, is that $P_i$, when given $y_{i-1}$, can efficiently compute $y_i$. Namely, let $y_{i-1} = f(x_1, \ldots, x_{i-1}, z_i, \ldots, z_n)$ be the residual truth-table after the first $i-1$ inputs $x_1, \ldots, x_{i-1}$ have been set. (The final output of the function depends on the values of the variables $z_i, \ldots, z_n$ which haven't yet been set.) $P_i$ knows the input $x_i$ and based on it is can compute $y_i$, the truth table after the first $i$ inputs are set. This is done by $P_i$ removing the first entry of the table $y_{i-1}$ if $x_i = 1$, or removing the the last entry of $y_{i-1}$ if $x_i = 0$.

As an example, consider the MAJORITY function over 3 inputs. The truth table of this function is $(0, 0, 1, 1)$. Sup-

pose that $P_1, P_2, P_3$ have inputs $0, 1, 0$, respectively. Then $y_0 = (0, 0, 1, 1)$, and subsequently $y_1 = (0, 0, 1)$, $y_2 = (0, 1)$, and $y_3 = (0)$. This last value is also the output of the function.

In [13] these observations about the decompositions of binary symmetric functions were also extended to symmetric functions over arbitrary domains. The size of the truth-tables of the latter functions is larger, and is $\binom{n+c-1}{n}$ for a domain of size $c$.

*Secure computation of function decompositions.*

A protocol is said to securely compute a given decomposition of $f$ if the only partial information that it leaks is the value of the decomposition after the last honest party provided its input. Namely, the view of any set of adversarial parties can be simulated knowing only the value $y_i = f_i(\ldots)$, where $i$ is the index of the last honest party. Furthermore, if the server is honest, then nothing but the final output of $f$ is revealed. This means that as long as there is an honest party that has not yet provided its input to the computation, the current intermediate value $y_i$ of the function can be hidden from the adversaries.

In [13], this notion of security is defined according to the ideal/real paradigm. The ideal model is defined for a specific function decomposition. The trusted party receives the inputs of all parties. If the set of corrupt parties does not include the server then they learn nothing. Otherwise, the trusted party gives to the corrupted server the output of the function as well as the value $y_i$, where $i$ is the index of the last honest party. Security is defined by requiring that the execution in the real model can be simulated given this information that is given in the ideal model to the corrupt server. Security is defined in both the semi-honest and malicious settings.

Protocols based on this security definition were designed in [13]. We describe them in Section 2.1.

### 1.2.2 Implementing cyrpto in the browser: Native Client technology

Native Client is an open-source technology that supports building web applications that seamlessly execute native compiled code inside the browser. Google implemented Native Client in the Chrome browser for running a subset of Intel x86 or ARM native code using software-based fault isolation. The goal is to maintain the OS portability and safety that people expect from web apps, while enjoying the performance of native code. Some of the techniques used by NaCl for sandboxing include restricting the memory range that the sandboxed code can access, using a code verifier to prevent unsafe instructions, in particular system calls, and requiring that all indirect jumps are to the start of 32-byte-aligned blocks. C code can be compiled to run under these constraints, using a compiler that is provided by the NaCl project. See more details in [20, 19, 1] and at `https://developers.google.com/native-client/overview`.

NaCl enables browsers to run code provided by web applications, at a speed that is only slightly slower than running native code. It currently supports C and C++ code, and is implemented in the Chrome browser. NaCl has been used for high-performance web based gaming, but it also seems ideal for MPC applications since these applications are computation intensive and need all the cpu power that they can get.

*Native Client compared to other alternatives.*

There are several options for providing web clients with code to run. They include (1) providing the client with a downloadable application, that the client must then install and run; (2) providing a Java applet that a browser can download in the form of bytecode and execute within the Java Virtual Machine (JVM); (3) providing javascript code that is run within the browser; and (4) providing NaCl code that is run in the browser.

We compare these alternatives and discuss our choice of NaCl technology in Section 3.1, where we also present measurements of the performance of these different alternatives. The conclusion is that NaCl technology provides the best mixture of usability, security, and performance, and should be the preferred choice for implementing complex cryptographic functionalities in the browser.

*Code verifiability.*

We note that it is straightforward to let users verify the correctness of the NaCl executable that they receive. All that is required is to make the C/C++ source code available for download. Interested users can then download the source code, inspect it to verify that it computes the right functionality, compile it using the standard NaCl compiler, and compare the result to the executable provided by the NaCl web server. This verification process is possible for other programming languages as well, but is particularly easy for NaCl, since the executable must be compiled using a single specific NaCl compiler, regardless of the architecture and operating system that the user uses.

*Google Native Client vs. other technologies.*

At this time Native Client technology is only supported by the Chrome browser. While this support is multi-platform, it does not seem that other browsers are about to embrace NaCl in the near future. In fact, Mozilla has a new alternative technology called OdinMonkey, which is an asm.js optimizer that lets developers compile C++ code using the Emscripten compiler and result in code that runs as efficiently as native code. We remark that our work does not depend specifically on NaCl. Rather, it demonstrates that the deployment and usage of MPC can greatly benefit if web applications are able to provide code that is run natively in clients. The specific technology that is used is of lesser importance. Moreover, future versions of Canon-MPC can compile its MPC client software using NaCl, OdinMonkey, or any other technology that will exist for this purpose. Each browser will be served the code that it supports best. The different client-side versions of the software can all interact, and enable participants who use different browsers to execute MPC protocols together.

## 1.3 Related Work

Our work is based on the work of Halevi et al. [13] which initiated the study of web based one-pass protocols, introduced the basic definitions, and described very efficient constructions for specific classes of functions, as well as generic constructions for arbitrary functions. The latter construction are less practical at this time as they use re-randomizable garbled circuits that were introduced by Gentry et al. for the purpose of multi-Hop homomorphic encryption [10]. An earlier related work is that of Choi et al. [6] that considered a setting where the parties can interact before receiving their

inputs, and then have to minimize online communication while maintaining full security.

The work of [13] was extended in [12], where very efficient one-pass protocols are described for more general classes of functions, such as branching programs and multivariate polynomials.

Finally, we mention the non-interactive two-party secure computation protocols of [15]. These protocols operate in a two-party, rather than a multi-party, setting. They enable a computation of any functionality in just a single round between the two parties.

# 2. THE PROTOCOL

We describe here the basic protocol presented in [13] for computing binary symmetric functions, as well as its modification to be secure against malicious adversaries. We also present a new improvement, of batching the zero-knowledge proofs in the malicious case. This change greatly improves the efficiency of the protocol in that setting, as is evident by our performance measurements in Section 3.3.

## 2.1 The HLP Protocol

We describe here the protocol of [13] that we implemented, which supports the secure computation of symmetric binary functions. This protocol was extended in [13] to support symmetric functions over arbitrary domains. That extension can be easily added to our implementation (but we have not coded it yet).

The protocol is run in the PKI model. Namely, before the protocol is run each party chooses a private-public key pair according to a known algorithm, and publishes the public key (i.e., registers it with the system web site). The parties are denoted $P_1, \ldots, P_n$, and have inputs $x_1, \ldots, x_n$, respectively. In addition, the server has a public key which we denote as $h_{n+1}$.

The protocol is based on the usage of El Gamal encryption over a group $\mathbb{G}$ of prime order $q$ with generator $G$. In our work we implemented the group $\mathbb{G}$ using operations in a subgroup of prime order of $Z_p^*$, namely using multiplication modulo a prime number $p$. The implementation could have also been based on arithmetic in an elliptic curve group, using known groups used for elliptic curve cryptography. This would have resulted in a more efficient implementation, in terms of both computation and the length of messages that have to be communicated. We leave this (simple) change to future work.

### *El Gamal encryption with multiple keys.*

The protocol is based on the usage of El Gamal encryption. Let $\mathbb{G}$ be a group of prime order $q$ with generator $G$. Each user $P_i$ has a private key $\alpha_i \in [1, q]$ and a public key $h_i = G^{\alpha_i}$. Normally, encryption with a public key $h$ is done as $E_{pk}(x) = (G^r, h^r \cdot x)$. We define encryption under public keys $h_1, \ldots, h_{n+1}$ as $(G^r, (H_{1,n+1})^r \cdot x)$, where $H_{1,n+1} = \prod_{j=1}^{n+1} h_j = G^{\sum_{j=1}^{n+1} \alpha_j}$. In general, define $H_{i,n+1} = \prod_{j=i}^{n+1} h_j = G^{\sum_{j=i}^{n+1} \alpha_j}$.

### *Gradual decryption and rerandomization.*

In the protocol, each user $P_i$ needs to decrypt the part of the encryption that is based on its key $h_i$, and rerandomize the result. The encrypted plaintext is revealed after all users perform this task. This gradual decryption and rerandomization is performed in the following way. $P_i$ is given $(u, v)$ where $u = G^r$ and $v = (H_{i,n+1})^r \cdot x$. It decrypts by computing

$$u' = u \quad \text{and} \quad v' = v \cdot u^{-\alpha_i}.$$

The result $(u', v')$ is a valid encryption of $x$ with randomness $r$, under public key $H_{i+1,n+1}$. This ciphertext is then rerandomized by $P_i$ computing $u'' = u' \cdot G^s$ and $v'' = v' \cdot (H_{i+1,n+1})^s$. The result is an encryption of the same plaintext $x$, under the key $H_{i+1,n+1}$, and using fresh randomness $(r + s)$.

### *The protocol.*

The description of the protocol and its proof of security appear in [13]. We describe here a version of the protocol that is slightly modified to our setting.

In the initialization step, the server constructs the truth-table of $f$. The table has $n + 1$ entries. The server encrypts each entry using the El Gamal encryption scheme and the key $H_{1,n+1}$ and fresh randomness. (Of course, the El Gamal scheme only encrypts values in the group $\mathbb{G}$. Therefore there must be some agreed representation for 0 by a value in $\mathbb{G}$.)

In Step i, party $P_i$ interacts with the server and receives from it a table with $n - i + 2$ entries, each encrypted with the key $H_{i,n+1}$. $P_i$ then performs the following operations

- If $x_i = 0$ it removes the last entry of the table. Otherwise ($x_i = 1$) it removes the first entry of the table.
- It decrypts all remaining entries of the table using its private key $\alpha_i$.
- It rerandomizes the encryption each entry of the table, and sends the resulting table to the server.[2]

The server receives from $P_i$ a table of length $n - i + 1$, where each entry is encrypted with the key $H_{i+1,n+1}$. Note that after the last step, performed by $P_n$, the server receives a table with a single entry that is encrypted with the key $H_{n+1,n+1} = h_{n+1}$. The server decrypts the table using its private key and reveals the output of the function.

The protocol works correctly regardless of the order in which the parties interact with the server, as long as all parties interact with the server. We discuss in Section 4 the case of absentee parties, namely parties whose keys were used for encrypting the table but who do not perform the step in which they should interact with the server and decrypt their layer of the encryption.

## 2.2 The Malicious Case

The basic protocol is secure only against semi-honest adversaries. It is vulnerable to malicious adversaries that can behave arbitrarily. For example, a malicious server can generate a truth table of a function different than $f$. Malicious participants can completely change the truth table, instead

[2]Rerandomization is needed in order to randomize the decryption process. Otherwise the server could learn the input of, say, $P_1$: The server generates the encryptions and can therefore identify the results of $P_1$ decrypting the table entries with its key. The server also receives these decrypted table entries from $P_1$. It can therefore identity which entry of the table was removed by $P_1$. Rerandomization prevents this attack.

of just removing one of its entries. We describe here a protocol from [13] that we implemented with some modifications, and which is secure against malicious adversaries.

The basic idea of the protocol is for the server and every participant to prove in zero-knowledge that they operated according to the rules of the semi-honest protocol. These proofs must be non-interactive, and are therefore implemented using the Fiat-Shamir heuristic. Furthermore, the proof generated by $P_i$ must be forwarded by the server to all parties $P_j$ with index $j > i$ when they interact with the server. For this purpose each party must have a digital signature key pair, and each proof must be signed by the party that generated it.

Computing and verifying these zero-knowledge proofs seems at first as a computationally expensive task. However, performance can be greatly improved, and be quite reasonable, based on two observations:

- Since all encryptions are based on the El Gamal scheme, all proofs can be translated to proofs that a certain tuple is a Diffie-Hellman tuple. (In short, to prove that a pair $(u, v)$ is an encryption of $x$ under public-key $h$, namely that $(u, v) = (G^r, h^r \cdot x)$, one can prove that the tuple $(G, h, u, v/x)$ is a Diffie-Hellman tuple. It is possible to prove in a similar way that a certain encryption is a rerandomization of another encryption.)

  There are known and efficient techniques, based on Sigma protocols, for proving that a tuple is Diffie-Hellman. Generating a proof requires two exponentiations, and verification takes four exponentiations. See [14], Chapter 6 for details.

- Each participant needs to prove that multiple tuples are all Diffie-Hellman. To speed up our implementation we introduced a batching technique that enables to prove all these statements together, with an overhead that is greatly improved compared to providing an individual proof for each statement. See details below.

### *The protocol.*

The modified protocol operates in the following way. In the initialization step, the server also computes and signs a proof that each entry in the encrypted truth table encrypts a representation of the same 0 or 1 value as in the original truth table of $f$. (As noted above, this statement reduces to a statement that a tuple is Diffie-Hellman.)

In Step $i$, $P_i$ receives from the server the signed proofs generated by the server and all previous participants, and verifies these proofs. Then, $P_i$ removes either the first or last table of the truth table, based on its input. It decrypts with its key the remaining table entries and rerandomizes them. Note that if $P_i$ removes the first entry of the table then it must prove that each entry $j$ in the new table is a rerandomized decryption of entry $j+1$ in the previous table. This statement translates to a Diffie-Hellman proof for each table entry. Similarly, if $P_i$ removes the last entry of the table it must prove that each entry $j$ in the new table is a rerandomized decryption of entry $j$ in the previous table. Overall, $P_i$ must therefore prove the OR of two sets of Diffie-Hellman proofs. This proof of a compound statement can be done using the techniques of Cramer et al. [7], at a cost similar to that of proving both set of statements. $P_i$ signs the proof and sends it to the server.

All proofs are computed using the Fiat-Shamir paradigm. Namely, the random challenge that is needed in the second step of the Sigma protocol is computed as a SHA1 hash of the message computed in the first step. The usage of the Fiat-Shamir paradigm means that the random oracle model must be used in the analysis. In [13] it was shown that this modified protocol is secure against malicious adversaries.

### 2.2.1 Batching proofs to improve efficiency

The straightforward way of proving that multiple tuples are all Diffie-Hellman tuples is to provide an independent proof for each tuple. We present here an alternative proof method that batches all statements to a single Diffie-Hellman proof.

The batching works the following way: Let the tuples be

$$\{(g, h, u_i, v_i)\}_{i=1}^{n}.$$

Let $\gamma_1, \ldots, \gamma_n$ be random $L$-bit values (that are chosen by the verifier, or, as in our system, are chosen by a hash function in the Fiat-Shamir heuristic). Then instead of proving each tuple separately it is possible to the prove that the tuple

$$(g, h, \Pi_{i=1}^{n}(u_i)^{\gamma_i}, \Pi_{i=1}^{n}(v_i)^{\gamma_i})$$

is a Diffie-Hellman tuple.

If all tuples are Diffie-Hellman then the new tuple is also Diffie-Hellman, regardless of the choice of the $\gamma$ values. Therefore completeness holds for the new proof. As for soundness, it is based on the following claim that was also stated in [18], Claim B.5:

CLAIM 2.1. *If there exists an index $1 \leq i \leq n$ such that $(g, h, u_i, v_i)$ is* not *a Diffie-Hellman tuple, then for every choice of $\gamma_1, \ldots, \gamma_{i-1}, \gamma_{i+1}, \ldots, \gamma_n$ there exists at most a single value $\gamma_i$ such that $(g, h, \Pi_{i=1}^{n}(u_i)^{\gamma_i}, \Pi_{i=1}^{n}(v_i)^{\gamma_i})$ is a Diffie-Hellman tuple.*

**Proof sketch:** Without loss of generality, assume that the $n$th tuple is not Diffie-Hellman,and assume also that $\gamma_1, \ldots, \gamma^{n-1}$ were already chosen, and now $\gamma_n$ is chosen at random. Then there is some value $a$ defined by the values $\gamma_1, \ldots, \gamma_{n-1}$ such that

$$\Pi_{i=1}^{n}(u_i)^{\gamma_i} = \Pi_{i=1}^{n-1}(u_i)^{\gamma_i} \cdot (u_n)^{\gamma_n} = g^a \cdot g^{\log_g(u_n)\gamma_n}.$$

Similarly, there is a value $b$ such that

$$\Pi_{i=1}^{n}(v_i)^{\gamma_i} = \Pi_{i=1}^{n-1}(v_i)^{\gamma_i} \cdot (v_n)^{\gamma_n} = g^b \cdot g^{\log_g(v_n)\gamma_n}.$$

For the tuple to be Diffie-Hellman it must hold that

$$a + \log_g(u_n)\gamma_n = b + \log_g(v_n)\gamma_n \bmod |\mathbb{G}|,$$

namely that

$$\gamma_n = (a - b)/(\log_g(v_n) - \log_g(u_n)) \bmod |\mathbb{G}|.$$

(This division is always possible since the tuple is not Diffie-Hellman and therefore $\log_g(v_n) \neq \log_g(u_n)$.)

The probability of setting $\gamma_n$ to the right value is at most $2^{-L}$. □

As a corollary, the soundness error of the protocol is at most $2^{-L}$ plus the soundness error of a single Diffie-Hellman proof.

*Efficiency improvement.*

The efficiency improvement is immense. The cost of proving and verifying $n$ proofs is reduced from $6n$ full exponentiations (due to a cost of 6 exponentiations per Diffie-Hellman proof), to computing one Diffie-Hellman proof and computing $4n$ exponentiations with the $\gamma_i$ exponents (due to each of the prover and verifier computing $2n$ exponentiations).

The parameter $L$, namely the length of the $\gamma_i$ exponents, can be set to be only 40 or 64 bits long (in our experiments we used $L = 64$). Therefore, the cost of exponentiations by the $\gamma_i$ exponents is about 40 or 64 divided by the order of the group in which the computations are done. The result is that batching a Diffie-Hellman statement is faster by an estimated factor between 3 or 4 (for $|L| = 64$ and the group being an elliptic curve group, to about 50 (for $|L| = 40$ and the group being $Z_p^*$ with $|p| = 2048$). An additional speedup comes from the fact only $4n$, rather than $6n$ exponentiations have to be computed. This is expected to reduce the overhead by an additional estimated factor of 2/3.

In our implementation we only implemented batched proofs, and therefore we do not have explicit measurements of the speedup from using batching compared to doing $n$ independent proofs. (See Section 3.3.) However, we observe there that due to the effect of batching the overhead of generating zero-knowledge proofs for the correctness of table manipulation is almost independent of the size of the table. See Tables 3 and 4 in Section 3.3.

# 3. THE SYSTEM

The system was developed as a web application, and is available for experimentation at `http://canon-mpc.org`.As is common with academic prototypes, the current code is suitable for experimentation and for measurements but not for running computations with sensitive data. We did not use TLS encryption, did not use secure coding, and the code might be vulnerable to SQL injections and other attacks. We describe here the major design choices, the system architecture, and the results of performance experiments that we conducted.

## 3.1 Design Choices

### 3.1.1 Providing the application to the user

As was stated earlier in the text, we chose to implement the client-side part of the system in Native Client. We describe here a comparison of the main alternatives for implementing the client side. The conclusion is that NaCl provides the best balance of properties for computation-intensive applications such as secure computation, and should therefore be used.

There are several options for providing web clients with the code that they are required to run. These options include (1) providing the client with a downloadable application, that the client must then install and run; (2) having the client launch a Java applet that is downloaded from a web page in the form of bytecode and executed within the Java Virtual Machine (JVM); (3) providing javascript code that is run within the browser; and (4) providing NaCl code that is run in the browser. We describe below the main properties of each of these options in terms of usability, security, and performance, and summarize these properties in Table 1.

| Technology | Usability | Security | Performance |
|---|---|---|---|
| Native applications | bad | bad | optimal |
| Java applets | mediocre | not good | mediocre |
| Javascript | best | good | bad |
| Native Client | good | good | almost optimal |

**Table 1: Comparison of different client side technologies**

*Downloadable native applications* offer bad usability, since users must manually download and install them. Furthermore, any update of the application requires all clients to download and install the most up to date version of the application. Another disadvantage is that the development of a multi-platform application is quite complicated, as it must be tailored to each specific architecture on which the application might be run. Applications are also a security nightmare, since they run natively on the client machine with system access, and are not isolated in any sandbox. On the other hand, native applications can use optimized code for the target machine and can provide the best performance.

*Java applets* that are downloaded by the browser from a web site offer mediocre usability, since Java has to be enabled in the browser, and the interaction between the applet and the web application is often not smooth. As for security, many security vulnerabilites were found in Java, and a recent CERT recommendation is "Unless it is absolutely necessary to run Java in web browsers, disable it" (`http://www.kb.cert.org/vuls/id/625617`). In terms of performance, native Java provides pretty good but not optimal performance for bignum operations. Better performance can be obtained by using the Java Native Interface (JNI), which enables the Java virtual machine to call native applications. However, it is not trivial to use JNI within applets.

*Javascript* offers great usability and seamless interaction with web applications. To ensure security, Javascript scripts are run in a sandbox in which they can only perform web-related actions, and are constrained by the same origin policy. Javascript vulnerabilites do exist, but are fewer than when using applets or downloaded applications. The performance of Javascript bignum operations is pretty bad, as is evident from Table 2. A main limiting factor for performance is memory management in a dynamic programming language.

*Native Client* technology has good usability. It offers seamless integration with web apps, but it is currently supported only in the Chrome browser. Its sandboxing security measures have proven so far to be pretty solid. In terms of performance, it is only slightly slower than running native applications (available data suggests a slowdown of between 5% to 30%).

We performed a comparison of the run time of public key operations in each of these technologies.[3] The results are summarized in Table 2. They include the run time of El Gamal encryption and decryption over $Z_p^*$ with $|p| = 1024$, and of the Damgård-Jurik additively homomorphic encryption scheme with a public key of length 2048 bits, as an example of an advanced crypto primitive [8]. The implementation in the native desktop application and in NaCl was done using C and the NTL bignum library. The Java imple-

[3]Of course, browsers also use cryptographic functions as part of the TLS protocol. However, it is impossible for web pages or web apps to access this functionality except for securing TLS traffic.

mentation used the Java BigInteger library. The Javascript implementation used the popular jsbn bignum library (at `http://www-cs-students.stanford.edu/~tjw/jsbn`). The Javascript versions that were used were SpiderMonkey version 17 for Firefox, and V8 version 3.9 for Chrome. All the measurements were performed on MacBook Pro, OS X 10.8.3, 2.5GHz Intel Core i5, 4GB memory.

Each result in the table is the average of 100 invocations of the function, where all random numbers used in the measurements were chosen in a preprocessing phase (to not interfere with the measurements). The results clearly demonstrate the performance advantage in using the native applications or NaCl, over the usage of Java or Javascript.

| Operation | Native app | NaCl | Chrome V8 | Firefox Spider Monkey | Java 1.6 |
|---|---|---|---|---|---|
| EG enc | 1.2ms | 1.5ms | 65ms | 726ms | 7.4ms |
| EG dec | 0.5ms | 0.71ms | 37ms | 782ms | 4.1ms |
| DJ enc | 5.8ms | 7.4ms | Not | | 93ms |
| DJ dec | 9.5ms | 10.3ms | Implemented | | 180ms |

**Table 2: Performance of cryptographic functions**

### 3.1.2 *Cryptographic library*

The current phase of the canon-mpc project is implemented using El Gamal encryption over $Z_p^*$. The intention is to change to encryption over elliptic curves in order to support larger computations (in particular, support computation of symmetric functions over arbitrary domains, which result in larger truth tables). We considered the usage of the following cryptographic libraries.

- NaCl – Networking and Cryptography Library (with no relation to the sandboxing technology of NaCl – Native Client), `http://nacl.cr.yp.to` [4]. This library is written in C++ and has excellent performance and great features, but is less flexible. It does not allow cryptographic primitives breakable in substantially fewer than $2^{128}$ operations, such as RSA-2048. It also takes advantage of low-level system architecture features and therefore cannot be exported to an ARM environment.

- SCAPI is an open-source Java library for implementing secure computation `http://crypto.biu.ac.il/scapi`. It uses JNI to encapsulate efficient big number implementations in other languages. However, due to the use of Java, it is close to impossible to use this library with Native Client.

- Crypto++ is a popular cryptographic library written in C++ `http://www.cryptopp.com`. However, it is non-trivial to compile crypto++ to non x86 architectures such as ARM, and in that case performance is affected since assembler usage must be disabled.

- NTL is a portable library for number theory `http://www.shoup.net/ntl/`. It is platform independent, reliable, and written entirely in C++. This library is therefore ideal for a NaCl application that will be used in different architectures.

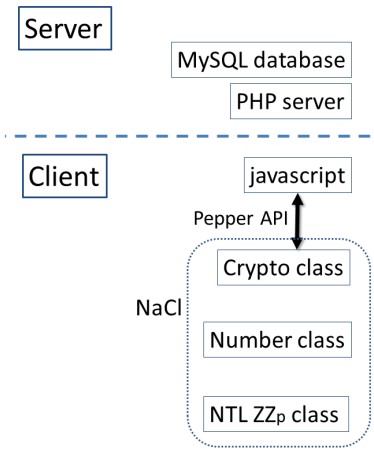


**Figure 1: System architecture.**

We chose to use the NTL library, and in particular its $ZZ_p$ class for integer computation modulo $p$. The cryptographic operations were implemented in $Z_p^*$ with a modulus of length $|p| = 1024$ bits, to achieve a reasonable level of security. Better security can be achieved by using a longer modulus of length $|p| = 2048$ or more, or, better still, using elliptic curves that offer comparable security with better efficiency.

## 3.2 Architecture

The architecture of the system is described in Figure 1.

*The client side.*

The client is implemented in javascript, which calls NaCl functions using the Pepper API (Pepper is an API for creating Native Client modules). It interacts with the `crypto` class, which is an API that we wrote for calling the different cryptographic functions that are implemented in NaCl. The `crypto` class interacts with the `numbers` class, which currently translates all calls to calls to the NTL library. When future lower level cryptographic libraries will be added to the system (say, the MIRACL library for elliptic curve math, or the Crypto++ class), the `numbers` class alone should be changed to support calls to these libraries. At the bottom level is the NTL $ZZ_p$ library, for performing integer computation modulo arbitrary integer numbers.

In our current implementation the NTL library was compiled without any optimizations, to achieve better portability and simplify the work with the NaCl compiler. It is possible to optimize performance by compiling for the x64 and x32 architectures, and enabling the use of gmp (the GNU Multi-Precision library), which uses assembly routines for a wide variety of architectures. Using these optimizations with the NaCl compiler is left for future work.

*The server side.*

The server side application was built using PHP. All data is stored in a MySQL server. The main page provides links to the following pages:

- A page where the user can experiment with running the different cryptographic primitives that were implemented in NaCl, and measure their performance.

- A "public parameters" page, which presents the public parameters used by the system for encryption. Currently it presents the modulus $p$, and a generator $g$ of $Z_p^*$.
- A registration page. Registration is required before users can participate in the computation (recall that the protocol is in the PKI model). During registration users provide information that is stored in the server's database, and includes an email address, a hash of the user's password, a public key and an encrypted private key.

  The actual registration process works by the client loading NaCl code which generates a key pair at the client machine. The private key is then encrypted using the password, and then the public key and the encrypted private key are sent to the server.
- A login page, where users are able to login to the system using their email address and password.
- A page where a user can initiate a protocol or participate in a protocol after receiving an invitation for that protocol. This page is described next.

*The Protocols page.*

The Protocols page supports running secure multi-party computation of symmetric binary functions as in [13]. The implemented protocols are secure against malicious adversaries.

A registered user can create and participate in any number of protocols. The user is shown a list of protocols in which he or she is invited to participate. Upon choosing a protocol to participate in, the user receives the current status of the protocol. Namely, the user is given the current encrypted truth-table and the proofs that were generated by the previous participants in the protocol. The user then verifies the proofs and is able to choose an input of either 0 or 1. The truth-table is updated, decrypted and rerandomized, according to the protocol that we described. In addition, the client generates a batched ZK proof proving that it operated according to the protocol. All computation is done in the client side and the results are sent to the server.

The page also enables any registered user to initiate a new protocol. The user must choose the registered users that will be invited to participate in the protocol. As a result, the public keys of these users are recovered from the database that is stored at the server and are sent to the user. The user then defines the function to be computed by entering a truth-table of appropriate length. Each row of the table is encrypted on the client side according to the protocol, using the public keys retrieved from the server. In addition, the client generates a ZK proof proving that it operated according to the protocol. The resulting encrypted truth-table and the proof are sent to the server. The server then emails the users who were invited to participate in the protocol and notifies them about the invitation.

## 3.3 Performance

This section describes the results of measurements of the running time of the different components of the client. All measurements were performed on MacBook Pro, OS X 10.8.3, 2.5GHz Intel Core i5, 4GB memory.

The code that was measured implemented the version of the protocol that is secure against *malicious* adversaries. (The running time for the semi-honest case can be easily extracted from the measurements, as is described below.) The code used El Gamal encryption, with a modulus of length 1024 bits that should provide reasonable security. The code implemented batching of the zero-knowledge proofs of Diffie-Hellman tuples, using exponents of length 64 bits.

We report the results of measurements of two instantiations of the protocol, the first with 5 participants and the second with 10 participants. All measurements were done to the NaCl code alone, since javascript measurements were less predictable due the asynchronous communication between javascript and NaCl. The reported results are the average of five runs.

*Receiving the NaCl code.*

Before the client can begin the computation, it must download the compiled NaCl code. The code turned out to be pretty large, at about 6MB, and therefore the download time was noticeable. We note however that Chrome caches NaCl code that it downloads, and therefore there is no need to download the code again if the protocol is run more than a single time.

*Generating keys.*

Generating a key pair at the client when registering to the system took only 5msec.

*Creating the truth-table.*

The initiator of the protocol generates an encrypted truth-table and proves its correctness. Creating a table for a computation of five participants took 125 msec. Generating a ZK proof of the correctness of this table took 530 msec. When instantiating the protocol for ten participants, the table generation took 400 msec, and the ZK proof took 802 msec.

*Participating in the protocol.*

Each participant in the protocol performs four tasks: (1) verifying the ZK proofs that it received; (2) randomizing and decrypting the table (after removing one entry from it); (3) proving in zero-knowledge that the randomization was correct; (4) proving in zero-knowledge that the decryption was correct.

We describe in Table 3 the run times of each of these steps for a computation with five participants, and describe in Table 4 the run times for a computation with ten participants. An immediate observation from the tables is that the total run time is good in terms of the user experience, and is less than one second for all users. The main components of the run time are the verification of ZK proofs, and the randomization and decryption of the tables.

The verification of the proofs received by the client takes longer for later participants, since they have to check the proofs generated by all previous participants. On the other hand, decrypting and rerandomizing the table becomes more efficient, since the size of the table shrinks.

The generation of new proofs takes considerably less time than the other tasks. It is interesting to note that the time it takes to generate proofs changes very little with the identity of the user, i.e. with the size of the table that the user has

| User | ZK verification | Randomize & decrypt | ZK proof of randomization | ZK proof of decryption | Total |
|---|---|---|---|---|---|
| 1 | 33 | 164 | 57 | 35 | 289 |
| 2 | 112 | 127 | 53 | 32 | 324 |
| 3 | 225 | 97 | 48 | 30 | 400 |
| 4 | 300 | 63 | 44 | 28 | 435 |
| 5 | 383 | 27 | | | 410 |

**Table 3: Running times in msec of the different tasks of a participant in a computation with 5 participants.**

| User | ZK verification | Randomize & decrypt | ZK proof of randomization | ZK proof of decryption | Total |
|---|---|---|---|---|---|
| 1 | 43 | 301 | 66 | 42 | 452 |
| 2 | 141 | 284 | 63 | 39 | 527 |
| 3 | 262 | 249 | 59 | 37 | 607 |
| 4 | 346 | 220 | 58 | 35 | 659 |
| 5 | 456 | 193 | 58 | 33 | 740 |
| 6 | 547 | 155 | 57 | 32 | 791 |
| 7 | 648 | 130 | 56 | 31 | 865 |
| 8 | 766 | 93 | 53 | 30 | 942 |
| 9 | 852 | 65 | 51 | 30 | 998 |
| 10 | 911 | 27 | | | 938 |

**Table 4: Running times in msec of the different tasks of a participant in a computation with 10 participants.**

to process. The size of the table affects the number of Diffie-Hellman statements that are included in the proof, but this is not noticeable due to the effect of batching.

### *The overhead in the semi-honest case.*

If the protocol only needs to be secure against semi-honest adversaries, then there is no need to generate or verify proofs. Therefore the overhead consists only of randomizing and decrypting the tables, i.e. of the third column in Tables 3 and 4. The resulting latency is much smaller than for the malicious case (e.g., less than 0.3sec in the ten participants case).

### *Improvements.*

The run time of the clients can be improved by using elliptic curve based El Gamal encryption. Doing this should provide improved security comparable or better than a 2048 bit modulus, at a smaller computation overhead. In addition, the size of the encrypted tables and of the proofs will also be reduced. The runtime can also be improved by turning on optimization options in the bignum library, and being able to use these compilation options within NaCl.

### *Summary and extensions.*

The run time that is reported here for the initial implementation of the system is quite good. The latency of the cryptographic operations is barely noticeable in terms of user experience, even for security against malicious adversaries. This is mainly due to the use of batching, which reduces the overhead of the ZK proofs to be almost independent of the table size.

The computation time might become too large if the number of participants is much bigger, or if we implement secure computation of symmetric functions over arbitrary domains. (The size of the table for that case is $\binom{n+c-1}{n}$ for a domain of size $c$, based on a protocol presented in [13].) For these usage scenarios it will be required to optimize the performance of the cryptographic primitives and use elliptic curve based encryption, as suggested above.

# 4. HANDLING ABSENTEE PARTICIPANTS

One practical obstacle to the deployment of the protocols of Halevi et al. [13] is that *all* users who are invited to participate in a computation *must* show up in order for the protocol to complete.

Suppose that you run a survey and invite ten of your friends to participate in it. Suppose also that one of the invitees cannot participate in the survey, or just doesn't care about it. After all other nine individuals have participated in the protocol you are left with a truth-table of size two, encrypted with the key of the absentee participant. It is possible to decide which default value to give to the missing input, and therefore know which entry to remove from the truth-table, but there is no way to decrypt the encryption that was performed with the public key of the missing participant.

It is preferable, of course, to have a one-pass protocol that can tolerate a number of absentee participants. Namely, let the initiator of the protocol define some deadline so that if not all invited participants actively participate in the protocol until that deadline, the server (or some other defined party) can recover the output of the function, where the inputs of all absentee participants are set to a default value, say to 0.[4] In that case the protocol must not, of course, reveal anything more than the output of the function. In addition, if all the invitees do participate in the protocol, then the normal security definitions should hold.

We show, however, that it is unlikely that such a protocol can be constructed without special assumptions about the setting in which it is run. We then show protocols that tolerate absentee participants and run in a single pass if all participants arrive, and otherwise run in two passes, where the second pass imposes minimal overhead on the parties.

## 4.1 The Unlikelihood of One-Pass Protocols

Suppose that there exists a one-pass protocol for computing a function $f(x_1, \ldots, x_n)$ of the inputs of $n$ parties $P_1, \ldots, P_n$, and that the protocol satisfies the following two limited requirements:

- If all parties participate in the protocol and none of these parties colludes with the server, then the value of $f(x_1, \ldots, x_n)$ is computed, and no other information is revealed to any party.

- If not all parties show up until some deadline, then the protocol computes the output of $f$ where the inputs $x_i$ of all absentee parties are set to 0. If no party colludes

[4]Setting the inputs of absentee participants to a default value seems natural. Indeed, many functions of $n$ inputs can be reduced to functions of $n' < n$ inputs by setting values to $n - n'$ inputs. For example, the majority function can be reduced in this way by setting half of the $n - n'$ missing inputs to 0 and setting the other half to 1.

with the server then no information except for this value of $f$ is revealed.

Note that the fact that the protocol is one-pass implies that the computation that satisfies the second requirement must be performed by the server, given the information that it already received and without contacting any of the parties that have already participated in the protocol (and, of course, without being able to contact the absentee parties).

Consider a setting where no party colludes with the server. The computation begins, and until some time before the deadline all parties except for $P_n$ participate in the protocol. Therefore the following two options must hold:

- If $P_n$ participates in the protocol before the deadline, then the server must compute $f(x_1, \ldots, x_n)$ and nothing else.
- If $P_n$ does not participate in the protocol until the deadline, then the server must be able to compute $f(x_1, \ldots, x_{n-1}, 0)$ and nothing else.

Suppose that $P_n$ participates in the protocol before the deadline. The server therefore learns $f(x_1, \ldots, x_n)$. But a corrupt server, even if it behaves in a semi-honest manner, can also use the information it learned from the other parties to compute by itself the value $f(x_1, \ldots, x_{n-1}, 0)$ and check if $f(x_1, \ldots, x_n) = f(x_1, \ldots, x_{n-1}, 0)$. If these two values are different, it deduces that $x_n = 1$. The server learns in this way more information than is allowed by the security definition. (This argument can extended to the case of more absentee participants.)

The above argument shows that a one-pass protocol tolerating absentees must use a different setting with a different set of assumptions. It is unclear to us how to construct such a protocol.

## 4.2 Optimistic Two-Pass Protocols Tolerating Absentees

This section describes an optimistic two-pass protocol tolerating absentees. The protocol is optimistic in the sense that if all invited parties participate in the protocol then the computation is completed in a single pass. Otherwise, the computation requires a second pass by some of the parties.

### 4.2.1 A solution based on threshold decryption

The problem with absentee participants is that the other participants are left with truth-table entries that are encrypted with the keys of these absentee participants and cannot be decrypted. The most straightforward solution is, therefore, to enable other parties to perform the decryption task, instead of each invited participant that does not show up for the computation. An existing tool for this purpose is threshold decryption: the private key of each party is divided to $n$ shares, such that any combination of $t$ of these shares enables decryption, and any usage of less than $t$ shares reveals no information about the ciphertext. Luckily, there are well known and tested threshold solutions, based on Shamir secret sharing, for El Gamal encryption [5].

The system is based on the Halevi et al. scheme that is implemented in Canon-MPC, and operates in the following way:

- Each party shares its private key among a set of semi-trusted parties, using a threshold scheme suitable for the El-Gamal encryption, with some predefined threshold $t$. One option is to share the key among the $n$ participants using a $t$-out-of-$n$ sharing. Another option is to share the key among a different set of designated parties or servers. The correctness of the sharing can be proven in zero-knowledge, or using verifiable secret sharing (vss) by the key owner, see, .e.g., [16, 5].
- If all parties show up for the computation then the protocol is carried out as usual.
- If some parties do not show up for the computation then a second round is executed. This round involves $t$ of the parties among which the shares were shared. These parties first identify the absentee parties by examining the signatures posted by the participants in the protocol. They then remove from the truth-table the rows that correspond to the default inputs set for the absentee participants. Finally, they use their shares to decrypt of the remaining encrypted row, resulting in the plaintext output of the protocol.

  This second phase of the protocol is also one-pass. The first party participating in it identifies the absentee parties, removes the relevant rows from the matrix, and performs its part of the decryption. Each other participant in this protocol phase, first verifies the operations performed by the previous participants and then performs its part of the decryption.

*Security.*

If an adversary controls less than $t$ of the parties that share the key, the protocol is as secure as the one-pass protocol.

THEOREM 4.1. *Let $f$ be a symmetric function and let the encryption scheme be layer rerandomizable. In addition, all the keys of the honest parties are generated and shared correctly between all $n$ parties, using $t$-out-of-$n$ secret sharing. Then, as long as at most $t-1$ parties collude, the protocol described here achieves the same privacy as the one-pass protocol of [13] described in Section 1.2.1.*

PROOF. (informal) Unlike the protocol of [13], each private key is shared using $t$-out-of-$n$ sharing. Any set of $t-1$ corrupt parties, receives shares that are truly random, and therefore these shares unconditionally reveal no information about the private key of any other participant.

We separate the remainder of the proof into two cases, (i) all parties participate in the first pass of the protocol; (ii) at least one party does not participate in the first pass of the protocol.

In the first case there is no need for the second round of the protocol. Namely, in this case the parties run the one-pass protocol of [13], which is proven to be secure as long as no information about the the keys is leaked (which is the case if at most $t-1$ parties collude).

In the second case, at least one party does not take part in computing $f$, and therefore $t$ parties run a second pass where they remove the remaining layers of encryption. If we assume the threshold decryption scheme to be secure, then this round simply implements the decryption that should have been performed in the first round, and is therefore secure. □

# 5. REFERENCES


[1] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. *SIGPLAN Not.*, 47(6):355–366, June 2011.

[2] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 257–266. ACM, 2008.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In J. Simon, editor, *STOC*, pages 1–10. ACM, 1988.

[4] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In A. Hevia and G. Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer-Verlag Berlin Heidelberg, 2012.

[5] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In M. J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 1999.

[6] S. G. Choi, A. Elbaz, T. Malkin, and M. Yung. Secure Multi-party Computation Minimizing Online Rounds. In M. Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 268–286. Springer, 2009.

[7] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols. In *Advances in Cryptology - CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994.

[8] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In K. Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[9] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[10] C. Gentry, S. Halevi, and V. Vaikuntanathan. i-Hop Homomorphic Encryption and Rerandomizable Yao Circuits. In *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2010. Full version available on-line from `http://eprint.iacr.org/2010/145`.

[11] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In A. V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.

[12] D. Gordon, T. Malkin, M. Rosulek, and H. Wee. Multi-party computation of polynomials and branching programs without simultaneous interaction. In *Eurocrypt*, 2013.

[13] S. Halevi, Y. Lindell, and B. Pinkas. Secure computation on the web: Computing without simultaneous interaction. In P. Rogaway, editor, *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 132–150. Springer, 2011.

[14] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, November 2010.

[15] Y. Ishai, E. Kushilevitz, R. Ostrovsky, M. Prabhakaran, and A. Sahai. Efficient non-interactive secure computation. In K. G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 406–425. Springer, 2011.

[16] S. Jarecki. *Efficient Threshold Cryptosystems*. PhD thesis, MIT, 2001.

[17] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure mpc with dishonest majority. *IACR Cryptology ePrint Archive*, 2013:143, 2013.

[18] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Y. Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[19] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[20] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, Jan. 2010.