Implementing Two-Party Computation Efficiently with Security Against Malicious Adversaries*

Yehuda Lindell[1], Benny Pinkas[2], and Nigel P. Smart[3]

[1] Dept. Of Computer Science,
Bar Ilan University,
Ramat Gan, Israel,
lindell@cs.biu.ac.il
[2] Dept. of Computer Science,
University of Haifa
Haifa 31905, Israel
benny@pinkas.net
[3] Dept. Computer Science,
University of Bristol,
Woodland Road, Bristol, BS8 1UB, United Kingdom,
nigel@cs.bris.ac.uk

Abstract. We present an implementation of the protocol of Lindell and Pinkas for secure two-party computation which is secure against malicious adversaries [13]. This is the first running system which provides security against malicious adversaries according to rigorous security definition and without using the random oracle model. We ran experiments showing that the protocol is practical. In addition we show that there is little benefit in replacing subcomponents secure in the standard model with those which are only secure in the random oracle model. Throughout we pay particular attention to using the most efficient subcomponents in the protocol, and we select parameters for the encryption schemes, commitments and oblivious transfers which are consistent with a security level equivalent to AES-128.

1 Introduction

Secure multi-party computation is a process which allows multiple participants to implement a joint computation that, in real life, may only be implemented using a trusted party. The participants, each with its own private input, communicate without the help of any trusted party, and can compute any function without revealing information about the inputs (except for the value of the function). A

* The first author was supported by The Israel Science Foundation (grant No. 781/07) and by an Infrastructures grant from the Israeli Ministry of Science. The other authors were supported by the European Union under the FP7-STREP project CACE. The second author was also supported by The Israel Science Foundation (grant No. 860/06).

classic example of such a computation is the Millionaires' problem, in which two millionaires want to know who is richer, without revealing their actual worth.

Multi-party computation has been considered by the theoretical cryptography community for a long time, starting with the pioneering work of Yao [24] in 1986. Yao's garbled circuit construction is relatively simple, and runs in a constant number of rounds. Yao's construction still remains the most attractive choice for generic secure two-party computation.

In recent years attention has focused on whether the theoretical work has any practical significance. In the two-party case the main contribution has been the FairPlay compiler [15], which is a generic tool translating functions written in a special high-level language to Java programs which execute a secure protocol implementing them. There are two major drawbacks with the current FairPlay implementation. Firstly it only provides weak security against malicious adversaries (where reducing the cheating probability to $1/k$ requires increasing the overhead by a factor of k), and has no proof of security (in particular, it is clear that it cannot be proven secure under simulation-based definitions). As such, its usage can only be fully justified for providing security against honest but curious (aka semi-honest) adversaries.[1] Secondly it does not make use of the latest and most efficient constructions of its various component parts.

In recent years the theoretical community has considered a number of ways of providing a variant of Yao's protocol which is secure against malicious adversaries. In the current paper we examine one of the more recent and efficient protocols for providing security for Yao's protocol against malicious adversaries, namely the protocol of Lindell and Pinkas [13] which is proved to be secure according to a standard simulation based definition, and as such can be securely used as a primitive in more complex protocols (see [8, Chapter 7], which in turn follows [6]).

Our work presents the following contributions:

- We provide an efficient implementation of the protocol of [13], which is secure against malicious adversaries. This is, to our best knowledge, the first implementation of a generic two-party protocol that is secure against malicious adversaries according to a standard simulation based definition. The implementation demonstrates the feasibility of the use of such protocols.
- We derive a number of optimizations and extensions to the protocol and to the different primitives that it uses. Unlike prior implementations we pay particular attention to using the most efficient constructions for the various components. For example we use elliptic curve based oblivious transfer protocols instead of finite field discrete logarithm based protocols.

[1] The cryptographic community denotes adversaries which can operate arbitrarily as "malicious". Semi-honest (or honest but curious) adversaries are supposed to follow the protocol that normal users are running, but they might try to gain information from the messages they receive in the protocol. It is, of course, easier to provide security against semi-honest adversaries.

– We also examine the difference between using protocols which are secure in the random oracle model (ROM) and protocols in the standard model.[2] Of particular interest is that our results show that there appears to be very little benefit in using schemes which are secure in the ROM as opposed to the standard model.[3]

1.1 Related Work

Research on security against malicious adversaries for computationally secure protocols started with the seminal GMW compiler [9]. As we have mentioned, we base our work on the protocol of [13], and we refer the reader to that work for a discussion of other approaches for providing security against malicious adversaries (e.g., [14, 11, 23]). We note that a simulation based proof of security (as in [13]) is essential in order to enable the use of a protocol as a building block in more complex protocols, while proving the security of the latter using general composition theorems like those of [6, 8]. This is a major motivation for the work we present in this paper, which enables efficient construction of secure function evaluation primitives that can be used by other protocols. (For example, the secure protocol of [2] for finding the k^{th} ranked element is based on invoking several secure computations of comparisons, and provides simulation based security against malicious adversaries if the invoked computations have a simulation based proof. Our work enables to efficiently implement that protocol.)

The first generic system implementing secure two-party computation was FairPlay [15], which provided security against semi-honest adversaries and limited security against malicious adversaries (see discussion above). FairPlayMP is a generic system for secure *multi-party* computation, which only provides security against semi-honest adversaries [3]. Another system in the multi-party scenario is SIMAP, developing a secure evaluation of an auction using general techniques for secure computation [5, 4]. It, too, supports only security against semi-honest adversaries.

1.2 Paper Structure

Section 2 introduces Yao's protocol for secure two-party computation, while Section 3 presents the protocol of [13] which is secure against malicious adversaries.

[2] A random oracle is a function which is modeled as providing truly random answers. This abstraction is very useful for proving the security of cryptographic primitives. However, given any specific implementation of a function (known to the users who use it), this assumption no longer holds. Therefore it is preferable to prove security in the standard model, namely without using any random oracle.

[3] This is surprising since for more traditional cryptographic constructions, such as encryption schemes or signature schemes, the random oracle constructions are almost always twice as efficient in practice compared to the most efficient standard model schemes known. Part of the reason for the extreme efficiency of our standard model constructions is our use of a highly efficient oblivious transfer protocol which reduces the amortized number of zero-knowledge proofs which are required to be performed.

Section 4 presents the different efficient sub-protocols that we used. Finally, Section 5 presents the results of our experiments.

2 Yao's Garbled Circuit

Two-party secure function evaluation makes use of the famous garbled circuit construction of Yao [24]. In this section we briefly overview the idea. Note, however, that the following basic protocol is not secure against malicious adversaries, which is why the advanced protocol in the next section is to be preferred. The basic idea is to encode the function to be computed via a Binary circuit and then to securely evaluate the circuit on the players' inputs.

We consider two parties, denoted as P_1 and P_2, who wish to compute a function securely. Suppose we have a simple Binary circuit consisting of a single gate, the extension to many gates given what follows is immediate. The gate has two input wires, denoted w_1 and w_2, and an output wire w_3. Assume that P_1 knows the input to wire w_1, which is denoted b_1, and that P_2 knows the input to wire w_2, which is denoted b_2. We assume that each gate has a unique identifier Gid (this is to enable circuit fan out of greater than one, i.e. to enable for the output wire of a gate to be used in more than one other gate). We want P_2 to determine the value of the gate on the two inputs without P_1 learning anything, and without P_2 determining the input of P_1 (bar what it can determine from the output of the gate and its own input). We suppose that the output of the gate is given by the function $G(b_1, b_2) \in \{0, 1\}$

Yao's construction works as follows. P_1 encodes, or garbles, each wire w_i by selecting two different cryptographic keys k_i^0 and k_i^1 of length t, where t is a computational security parameter which suffices for the length of a symmetric encryption scheme. In addition to each wire it associates a random permutation π_i of $\{0, 1\}$. The garbled value of the wire w_i is then represented by the pair $(k_i^{b_i}, c_i)$, where $c_i = \pi_i(b_i)$.

An encryption function $E^s_{k_1,k_2}(m)$ is selected which has as input two keys of length t, a message m, and some additional information s. The additional information s must be unique per invocation of the encryption function (i.e., used only once for any choice of keys). The precise encryption functions used are described in Section 4.1. The gate itself is then replaced by a four entry table indexed by the values of c_1 and c_2, and given by

$$c_1, c_2 : E^{\mathrm{Gid}\|c_1\|c_2}_{k_1^{b_1},k_2^{b_2}} \left(k_3^{G(b_1,b_2)} \| c_3 \right),$$

where $b_1 = \pi_1^{-1}(c_1)$, $b_2 = \pi_2^{-1}(c_2)$, and $c_3 = \pi_3(G(b_1, b_2))$. Note that each entry in the table corresponds to a combination of the values of the input wires, and contains the encryption of the garbled value corresponding to these values of the input wires, and the corresponding c value. The resulting look up table (or set of look up tables in general) is called the Garbled Circuit.

P_1 then sends to P_2 the garbled circuit, its input value $k_1^{b_1}$, the value $c_1 = \pi_1(b_1)$, and the mapping from the set $\{k_3^0, k_3^1\}$ to $\{0, 1\}$ (i.e. the permutation

π_3). P_1 and P_2 engage in an oblivious transfer (OT) protocol so that P_2 learns the value of $k_2^{b_2}, c_2$ where $c_2 = \pi_2(b_2)$. P_2 can then decrypt the entry in the look up table indexed by (c_1, c_2) using $k_1^{b_1}$ and $k_2^{b_2}$; this will reveal the value of $k_3^{G(b_1,b_2)} \| c_3$ and P_2 can determine the value of $G(b_1, b_2)$ by using the mapping π_3^{-1} from the set c_3 to $\{0, 1\}$.

In the general case the circuit consists of multiple gates. P_1 chooses random garbled values for all wires and uses them for constructing tables for all gates. It sends these tables (i.e., the garbled circuit) to P_2, and in addition provides P_2 with the garbled values and the c values of P_1's inputs, and with the permutations π used to encode the *output* wires of the circuit. P_2 uses invocations of oblivious transfer to learn the garbled values and c values of its own inputs to the circuits. Given these values P_2 can evaluate the gates in the first level of the circuit, and compute the garbled values and the c values of the values of their output wires. It can then continue with this process and compute the garbled values of all wires in the circuit. Finally, it uses the π permutations of the output wires of the circuit to compute the real output values of the circuit.

Traditionally, for example in hardware design, one uses circuits which are constructed of simple gates which take at most two inputs and produce as most one output. In a Yao circuit a gate which takes n inputs and produces m outputs is encoded as a look up table which has 2^n rows, each consisting of a string of $O(m \cdot t)$ bits (where t is the security parameter which denotes the length of a key). Hence, it is often more efficient to use non-standard gates in a Yao circuit construction. For example a traditional circuit component consisting of k 2-to-1 gates, with n input and m output wires can be more efficiently encoded as a single n-to-m gate if $4k > 2^n$. In what follows we therefore assume the more suitable n-to-m gate construction. The extension of the above gate description to this more general case is immediate.

3 The Lindell-Pinkas Protocol

The protocol was presented in [13] and was proved there to be secure according to the real/ideal-model simulation paradigm [6, 8]. The proof is in the standard model, with no random oracle model or common random string assumptions. We describe below the protocol in some detail, for full details see [13]. We remark that this description is not essential in order to understand the results of our paper. The important things to note are the basic structure of the protocol, as described in the next paragraph, and the fact that the protocol is based on the use of different types of commitments (statistically binding, statistically hiding, and computational), and of an oblivious transfer protocol. We describe the implementation of these primitives in Section 4.

The basic structure of the protocol: The protocol proceeds in the following steps. It has statistical security parameters s_1 and s_2. We replace P_2's input wires with a new set of $O(s_2)$ input wires, and change the original circuit by adding to it a new part which translates the values of the new input wires to those of

the original wires. Then P_1 generates s_1 copies of Yao circuits and passes them to P_2, along with $O(s_1^2)$ commitments to the inputs. The input decommitments for P_1's inputs are transferred to P_2 via a batched oblivious transfer. Finally, after executing a number of cut-and-choose checks on the transferred circuits and commitments, P_2 evaluates half of the circuits and determines the output value as the majority value of the outputs of these circuits. One of the contributions of this paper is to examine each of the above operations in turn and optimize the parameters and components used in the Lindell-Pinkas description.

3.1 The Protocol in Detail

As explained in [13] it suffices to present a protocol for the case where the output is learnt by P_2 and P_1 learns nothing. We consider the computation of $f(x, y)$ where P_1's input is $x \in \{0,1\}^n$ and P_2's input is $y \in \{0,1\}^n$.

The protocol is parameterized by two statistical security parameters s_1 and s_2. (In [13] these are a single statistical security parameter but we shall see later that in order to optimize performance these parameters really need to be treated separately.) The protocol takes as input a circuit description $C^0(x, y)$ which describes the function $f(x, y)$. We use the notation com_b to refer to a statistically binding commitment scheme, com_h to refer to a statistically hiding commitment scheme, and com_c to refer to a commitment scheme which is only computationally binding and hiding. See Section 4 for our precise choice of these protocols.

The protocol itself is quite elaborate, but, as demonstrated in Section 5, it can be implemented quite efficiently.

0. CIRCUIT CONSTRUCTION: The parties replace C^0, in which P_2 has n input wires, with a circuit C in which P_2 has ℓ input wires, where $\ell = \max(4n, 8s_2)$. The only difference between the circuits is that each original input wire of P_2 in C^0 is replaced with an internal value which is computed as the exclusive-or of a random subset of the ℓ input wires of C. (Given an input to the original circuit, P_2 should therefore choose a random input to the new circuit, subject to the constraint that the internal values are equal to the original input values.) The exact construction is presented in Section 5.2 of [13]. (In order to avoid unnecessary extra gates in the circuit segment that computes the original input wires as a function of the new input wires, we designed the exact wiring using a variant of structured Gaussian elimination.)
   We let the new input wires of P_2 be given by $\hat{y} \leftarrow \hat{y}_1, \ldots, \hat{y}_\ell$
1. COMMITMENT CONSTRUCTION: P_1 *constructs the circuits and commits to them, as follows:*[4]
   (a) P_1 constructs s_1 independent copies of a garbled circuit of C, denoted $GC_1, \ldots, GC_{s_1}$.

[4] In [13] this commitment is done with a perfectly binding commitment scheme, however one which is computationally binding will suffice to guarantee security.

(b) P_1 commits to the garbled values of the wires corresponding to P_2's input to each circuit. That is, for every input wire i corresponding to an input bit of P_2, and for every circuit GC_r, P_1 computes the ordered pair

$$(c^0_{i,r}, c^1_{i,r}) \leftarrow (\mathrm{com}_c(k^0_{i,r}), \mathrm{com}_c(k^1_{i,r})),$$

where $k^b_{i,r}$ is the garbled value associated with b on input wire i in circuit GC_r. We let $(dc^0_{i,r}, dc^1_{i,r})$ denote the associated decommitment values.

(c) P_1 computes commitment-sets for the garbled values that correspond to its own inputs to the circuits. That is, for every wire i that corresponds to an input bit of P_1, it generates s_1 pairs of commitment sets $\{W_{i,j}, W'_{i,j}\}_{j=1}^{s_1}$, in the following way:
Denote by $k^b_{i,r}$ the garbled value that was assigned by P_1 to the value $b \in \{0,1\}$ of wire i in GC_r. Then, P_1 chooses $b \leftarrow \{0,1\}$ and computes

$$W_{i,j} \leftarrow \langle \mathrm{com}_c(b), \mathrm{com}_c(k^b_{i,1}), \ldots, com_c(k^b_{i,s_1})\rangle,$$
$$W'_{i,j} \leftarrow \langle \mathrm{com}_c(1-b), \mathrm{com}_c(k^{1-b}_{i,1}), \ldots, \mathrm{com}_c(k^{1-b}_{i,s_1})\rangle.$$

There are a total of $n \cdot s_1$ commitment-sets (s_1 per input wire). We divide them into s_1 *supersets*, where superset S_j is defined to be the set containing the jth commitment set for all wires. Namely, it is defined as $S_j = \{(W_{k,j}, W'_{k,j})\}_{k=1}^{n}$.

2. Oblivious transfers: For every input bit of P_2, parties P_1 and P_2 run a 1-out-of-2 oblivious transfer protocol in which P_2 receives the garbled values for the wires that correspond to its input bit (in every circuit).
Let $i_1, \ldots, i_w$ be the input wires that correspond to P_2's input, then, for every $j = 1, \ldots, w$, parties P_1 and P_2 run a 1-out-of-2 OT protocol in which:
   (a) P_1's input is the pair of vectors $[dc^0_{i_j,1}, \ldots, dc^0_{i_j,s_1}]$, and $[dc^1_{i_j,1}, \ldots, dc^1_{i_j,s_1}]$.
   (b) P_2's input are the bits $\hat{y}_j$, and its output should be $[dc^{\hat{y}_j}_{i_j,1}, \ldots, dc^{\hat{y}_j}_{i_j,s_1}]$.
3. Send circuits and commitments: P_1 sends to P_2 the garbled circuits, as well as all of the commitments that it prepared above.
4. Prepare challenge strings:[5]
   (a) P_2 chooses a random string $\rho_2 \leftarrow \{0,1\}^{s_1}$ and sends $\mathrm{com}_h(\rho_2)$ to P_1.
   (b) P_1 chooses a random string $\rho_1 \in \{0,1\}^{s_1}$ and sends $\mathrm{com}_h(\rho_1)$ to P_2.
   (c) P_2 decommits, revealing ρ_2.
   (d) P_1 decommits, revealing ρ_1.
   (e) P_1 andP_2 set $\rho \leftarrow \rho_1 \oplus \rho_2$.
   The above steps are run a second time, defining an additional string ρ'.
5. Decommitment phase for check-circuits: We refer to the circuits for which the corresponding bit in ρ is 1 as check-circuits, and we refer to the other circuits as evaluation-circuits. Likewise, if the jth bit of ρ' equals 1, then all commitments sets in superset $S_j = \{(W_{i,j}, W'_{i,j})\}_{i=1}^{n}$ are referred to as check-sets; otherwise, they are referred to as evaluation-sets.
   For every *check-circuit* GC_r, party P_1 operates in the following way:

[5] In [13] it is proposed to use perfectly binding and computationally hiding commitments here, but statistically binding and computationally hiding commitments actually suffice.

(a) For every input wire i corresponding to an input bit of P_2, party P_1 decommits to the pair $(c^0_{i,r}, c^1_{i,r})$.
(b) For every input wire i corresponding to an input bit of P_1, party P_1 decommits to the appropriate values in the *check-sets* $\{W_{i,j}, W'_{i,j}\}$.

For every pair of *check-sets* $(W_{i,j}, W'_{i,j})$, party P_1 decommits to the first value in each set i.e., to the value that is supposed to be a commitment to the indicator bit, com(0) or com(1)).

6. DECOMMITMENT PHASE FOR P_1'S INPUT IN EVALUATION-CIRCUITS: P_1 decommits to the garbled values that correspond to its inputs in the evaluation-circuits.
7. CORRECTNESS AND CONSISTENCY CHECKS: Player P_2 performs the following checks; if any of them fails it aborts.
   (a) *Checking correctness of the check-circuits:* P_2 verifies that each check-circuit GC_i is a garbled version of C.
   (b) *Verifying P_2's input in the check-circuits:* P_2 verifies that P_1's decommitments to the wires corresponding to P_2's input values in the check-circuits are correct, and agree with the logical values of these wires (the indicator bits). P_2 also checks that the inputs it learned in the oblivious transfer stage for the check-circuits correspond to its actual input.
   (c) *Checking P_1's input to evaluation-circuits:* Finally, P_2 verifies that for every input wire i of P_1 the following two properties hold:
      i. In every evaluation-set, P_1 chooses one of the two sets and decommitted to all the commitments in it which correspond to evaluation-circuits.
      ii. For every evaluation-circuit, all of the commitments that P_1 opened in evaluation-sets commit to the same garbled value.
8. CIRCUIT EVALUATION: If any of the above checks fails, P_2 aborts and outputs $\perp$. Otherwise, P_2 evaluates the evaluation circuits (in the same way as for the semi-honest protocol of Yao). It might be that in certain circuits the garbled values provided for P_1's inputs, or the garbled values learned by P_2 in the OT stage, do not match the tables and so decryption of the circuit fails. In this case P_2 also aborts and outputs $\perp$. Otherwise, P_2 takes the output that appears in most circuits, and outputs it.

3.2 The Statistical Security Parameters

The protocol uses two statistical security parameters, s_1 and s_2. The parameter s_1 is mainly used to prevent P_1 from changing the circuit that is evaluated, or providing inconsistent inputs to different copies of the circuit. The protocol requires P_1 to provide s_1 copies of the garbled circuit, and provide $(s_1)^2$ commitments for each of its input bits. The security proof in [13] shows that a corrupt P_1 can cheat with a success probability that is exponentially small in s_1. The original proof in [13] bounds the cheating probability at $2^{-s_1/17}$, which would require a large value of s_1 in order to provide a meaningful security guarantee. We conjecture that a finer analysis can provide a bound of $2^{-s_1/4}$, and in the full version of this paper we intend to prove this; this conjecture is based on an

analysis of a similar problem that was shown in [10]. A bound of $2^{-s_1/4}$ would mean that a relatively moderate value of s_1 can be used.[6]

The parameter s_2 is used to prevent a different attack by P_1, in which it provides corrupt values to certain inputs of the oblivious transfer protocol and then uses P_2's reaction to these values to deduce information about P_2's inputs (see [13] for details). It was shown that setting the number of new inputs to be $\ell = max(4n, 8s_2)$ bounds the success probability of this type of attack by 2^{-s_2}. The values of s_1 and s_2 should therefore be chosen subject to the constraint that the total success probability of a cheating attempt, $\max(2^{-s_1/4}, 2^{-s_2})$, is acceptable. Therefore, one should set $s_1 = 4s_2$.

3.3 Optimizing the Protocol Components

The protocol uses many components, which affect its overall overhead. These include the encryption scheme, the commitment schemes, and oblivious transfer. Much of our work was concerned with optimizing these components, in order to improve the performance of the entire protocol. We describe in the next section the different optimizations that we applied to the different components.

4 Subprotocols

To implement the above protocol requires us to define a number of sub-protocols: various commitment schemes, OT protocols and encryption schemes. In what follows we select the most efficient schemes we know of, in both the random oracle model (ROM) and the standard model. We assume that the concrete computational security parameter (as opposed to the statistical security parameter) is given by t. By this we mean that we select primitives which have security equivalent to t bits of block cipher security. Thus we first select an elliptic curve E of prime order $q \approx 2^{2t}$, and a symmetric cryptographic function with a t-bit key.

Elliptic curve. We let $\langle P \rangle = \langle Q \rangle = E$, an elliptic curve of prime order $q \approx 2^{2t}$, where no party knows the discrete logarithm of Q with respect to P.

Symmetric cryptographic function. The function that will be used for symmetric key cryptography is defined as a key derivation function $\mathrm{KDF}(m, l)$, which takes an input string m and outputs a bit string of length l. We use the KDF defined in ANSI X9.63, which is the standard KDF to use in the elliptic curve community [19]. It is essentially implemented as encryption in CTR mode where the encryption function is replaced by the SHA-1 hash function.

[6] The experiments in Section 5 assume a bound of $2^{-s_1/4}$. The overhead of different parts of the protocol is either linear or quadratic in s_1. If we end up using a worse bound of $2^{-s_1/c}$, where $4 < c \leq 17$, the timings in the experiments will be increased by factor in the range $c/4$ to $(c/4)^2$.

4.1 Encryption Scheme for Garbled Circuits

The encryption scheme $E^s_{k_1,k_2}(m)$ used to encrypt the values in the Yao circuit is defined by the algorithms in Figure 1. We assume that $k_i \in \{0,1\}^t$. The ROM version is secure on the assumption that the function KDF is modelled as a random oracle, whilst the standard model scheme is secure on the assumption that $\mathrm{KDF}(k\|s,l)$ is a pseudo-random function, when considered as a function on s keyed by the key k. We remark that the encryption is secure as long as the string s is used only once for any choice of key k. Note that the non-ROM version requires two invocations of the KDF, since we do not know how to analyze the security of a pseudo-random function if part of its key is known to an adversary (namely, if we use $\mathrm{KDF}(k_1\|k_2\|s,|m|)$, where KDF is modeled as a pseudo-random function, k_2 is secret and k_1 is known to an adversary, we cannot argue that the output is pseudo-random).

Figure 1 ROM and non-ROM encryption algorithms for the Yao circuits

Input: Keys k_1, k_2 of length t, and a string s. For encryption an l-bit message m in also given. For decryption, an l-bit ciphertext c is given.

ROM Version

Encryption $E^s_{k_1,k_2}(m)$

1. $k \leftarrow \mathrm{KDF}(k_1\|k_2\|s,|m|)$.
2. $c \leftarrow k \oplus m$.

Decryption

1. $k \leftarrow \mathrm{KDF}(k_1\|k_2\|s,|m|)$.
2. $m \leftarrow k \oplus c$.
3. Return m.

Non-ROM Version

Encryption $E^s_{k_1,k_2}(m)$

1. $k \leftarrow \mathrm{KDF}(k_1\|s,|m|)$.
2. $k' \leftarrow \mathrm{KDF}(k_2\|s,|m|)$.
3. $c \leftarrow m \oplus k \oplus k'$

Decryption

1. $k \leftarrow \mathrm{KDF}(k_1\|s,|c|)$.
2. $k' \leftarrow \mathrm{KDF}(k_2\|s,|c|)$.
3. $m \leftarrow c \oplus k \oplus k'$.
4. Return m.

4.2 Commitment Schemes

Recall we have three types of commitment schemes; statistically binding, statistically hiding and computationally binding/hiding, to commit to a value $m \in \{0,1\}^t$. (Note that the elliptic curve E is of order $q \approx 2^{2t}$ and so we can view m as a number in $\mathbb{Z}_q$ if desired.)

A Statistically Binding Commitment : $\mathbf{com_b}(m)$
We define the statistically binding commitment scheme as in Figure 2. The random oracle model based scheme is statistically binding, since to break the binding

property we need to find collisions in the hash function H. Since H is modelled as a random oracle, the probability of any adversary finding a collision given a polynomial number of points of H is negligible, even if it is computationally unbounded. The scheme is also computationally hiding by the fact that H is modelled as a random oracle (in fact, it's even statistically hiding if the adversary is limited to a polynomial number of points of H). The non-ROM scheme is statistically binding because P and c_1 fully determine r, which together with Q and c_2 in turn fully determine m. The fact that it is computationally hiding follows directly from the DDH assumption over the elliptic curve used.

Figure 2 ROM and non-ROM statistically binding commitment schemes

ROM Version
H is a hash function modeled as a random oracle.
Commitment $\text{com}_b(m)$

1. $r \leftarrow \{0,1\}^t$.
2. $c \leftarrow H(m\|r)$.
3. Return c.

Decommitment

1. Reveal m and r.
2. Check if $c = H(m\|r)$.
3. Return m.

Non-ROM Version
P and Q are elements on an elliptic curve, as described above.
Commitment $\text{com}_b(m)$

1. $r \leftarrow \mathbb{Z}_q$.
2. $c_1 \leftarrow [r]P$.
3. $c_2 \leftarrow [r][m]Q$.
4. Return (c_1, c_2).

Decommitment

1. Reveal m and r.
2. Check if $c_1 = [r]P$.
3. Check if $c_2 = [r][m]Q$.
4. Return m.

The Statistically Hiding Commitment : $\text{com}_h(m)$
For the statistically hiding commitment scheme we use the Pederson commitment [18]:

$$\text{com}_h(m) \leftarrow [r]P + [m]Q$$

where r is a random number of size q and we treat m as an integer modulo q. Note that $0 \leq m < 2^t < q < 2^{2t}$. Decommitment is performed by revealing r and m, and then verifying the commitment is valid. This is actually a perfectly hiding commitment (since given $\text{com}_h(m)$ there exists, for any possible value of m', a corresponding value r' for which $\text{com}_h(m) = [r']P + [m']Q$) and so in particular the commitment is also statistically hiding. That the commitment is computationally binding follows from the fact that any adversary who can break the binding property can determine the discrete logarithm of Q with respect to P.

A Computational Commitment Scheme : $\text{com}_c(m)$
We use the ROM version of the statistically binding commitment scheme in

Figure 2 for both the ROM and non-ROM commitments here. This is clearly suitable in the ROM. Regarding the non-ROM case, this scheme is computationally binding on the assumption that H is collision-resistant. Furthermore, it is computationally hiding when $H(m\|r)$ is modelled as a PRF with key r and message m. We remark that when m is large, this latter assumption clearly does not hold for typical hash functions based on the Merkle-Damgård paradigm (where given $H(k\|m)$ one can easily compute $H(k\|m\|m')$ for some m'). However, it is reasonable when m fits into a single iteration of the underlying compression function (as is the case here where $m \in \{0,1\}^t$ and t is a computational security parameter which we set to the value $t = 128$.).

4.3 Oblivious Transfer

Recall in our main protocol we need to perform $w = \max(4n, 8s_2)$ 1-out-of-2 oblivious transfers in Stage 2. We batch these up so as to perform all the OT's in a single batch. The OT's need to be performed in a manner which has a simulation based proof of security against malicious adversaries, hence the simple protocols of [17, 1, 12] are not suitable for our purposes (the simulation based proof is needed in order to be able to use a composition of the OT protocol in our protocol, see [6]). We therefore use a modification of the batched version of the protocol of Hazay and Lindell [10], which we now describe in the elliptic curve setting. (We note that this protocol has a simulation based proof of security in the standard model, without any usage of a random oracle.)

We assume that P_1's input is two vectors of values

$$[x_1^0, \ldots, x_w^0] \text{ and } [x_1^1, \ldots, x_w^1],$$

where $|x_j^0| = |x_j^1|$. Party P_2 has as input the bits $i_1, \ldots, i_w$ and wishes to obtain the vector $[x_1^{i_1}, \ldots, x_w^{i_w}]$.

We assume two zero-knowledge proofs-of-knowledge protocols which we shall describe in Appendix A. The first, $DL([x]P; x)$, proves, in zero-knowledge, knowledge of the discrete logarithm x of $[x]P$; the second, $DH(P, [a]P, [b]P, [ab]P)$, proves that the tuple P,$[a]P$,$[b]P$,$[ab]P$ is a Diffie–Hellman tuple.

The protocol follows. The main things to notice are that the zero-knowledge proofs of knowledge are performed only once, regardless of the number of items to be transfered, and that protocol is composed of only two rounds (in addition to the rounds needed by the zero-knowledge proofs).

1. P_2 chooses $\alpha_0, \alpha_1 \in \mathbb{Z}_q$ and computes $Q_0 \leftarrow [\alpha_0]P$ and $Q_1 \leftarrow [\alpha_1]P$, it then executes the protocol $DL(Q_0; \alpha_0)$ with party P_1.
2. For $j = 1, \ldots, w$ party P_2 chooses $r_j \in \mathbb{Z}_q$ and computes $U_j \leftarrow [r_j]P$, $V_{0,j} \leftarrow [r_j]Q_0 + [i_j]P$, $V_{1,j} \leftarrow [r_j]Q_1 + [i_j]P$. These values are then sent to P_1.
3. P_1 chooses $\rho_j \in \mathbb{Z}_q$, for $j = 1, \ldots, w$ and sends them to P_2.

4. Both parties then locally compute

$$U \leftarrow \sum_{j=1}^{w} [\rho_j] U_j, \quad V \leftarrow \sum_{j=1}^{w} [\rho_j](V_{0,j} - V_{1,j}).$$

Party P_2 executes the protocol $DH(P, Q_0 - Q_1, U, V)$ with party P_1.

5. For $j = 1, \ldots, w$ P_1 then performs the following steps:
   (a) Select $R_{0,j}, R_{1,j} \in \langle P \rangle$ at random.
   (b) Select $s_{0,j}, t_{0,j}, s_{1,j}, t_{1,j} \in \mathbb{Z}_q$.
   (c) Set $e_{0,j} \leftarrow (W_{0,j}, Z_{0,j}, y_{0,j})$ where

$$\begin{aligned} W_{0,j} &\leftarrow [s_{0,j}]U + [t_{0,j}]P, \\ Z_{0,j} &\leftarrow [s_{0,j}]V_0 + [t_{0,j}]Q_0 + R_{0,j}, \\ y_{0,j} &\leftarrow x_j^0 \oplus \mathrm{KDF}(R_{0,j}, |x_j^0|). \end{aligned}$$

   (d) Set $e_{1,j} \leftarrow (W_{1,j}, Z_{1,j}, y_{1,j})$ where

$$\begin{aligned} W_{1,j} &\leftarrow [s_{1,j}]U + [t_{1,j}]P, \\ Z_{1,j} &\leftarrow [s_{1,j}](V_1 - P) + [t_{1,j}]Q_1 + R_{1,j}, \\ y_{1,j} &\leftarrow x_j^1 \oplus \mathrm{KDF}(R_{1,j}, |x_j^1|). \end{aligned}$$

   The values $(e_{0,j}, e_{1,j})$ are then sent to P_2 for each value of j.

6. For $j = 1, \ldots, w$, party P_2 then computes

$$R \leftarrow Z_{i_j,j} - [\alpha_{i_j}] W_{i_j,j}$$

and outputs

$$x_j^{i_j} \leftarrow y_{i_j,j} \oplus \mathrm{KDF}(R, |x_j^{i_j}|).$$

For each index in the vector of inputs, the protocol requires P_1 to perform 10 multiplications, and P_2 to perform 8 multiplications. (This is without considering the zero-knowledge proofs, which are performed once in the protocol.) The security of the above scheme is fully proven in [10], with the only exception that here a KDF is used to derive a random string in order to mask (i.e., encrypt) the x_j^0 and x_j^1 values (in [10] it is assumed that x_j^0 and x_j^1 can be mapped into points in the Diffie-Hellman group). The use of a KDF for this purpose was proven secure in the context of hashed ElGamal in [22], on the assumption that KDF is chosen from a family of hash functions which are entropy smoothing.

5 Timings

In our implementation we selected $t = 128$ as the security parameter. As a result, we chose the KDF to be implemented by SHA-256, and as the elliptic curve E we selected the curve *secp256r1* from the SECG standard [20].

We performed a set of experiments which examined the system using a circuit which evaluates the function $x > y$ for inputs x and y of $n = 16$ bits in length.

The standard circuit (using simple 2-to-1 gates) for this problem consists of 61 2-to-1 gates and 93 internal wires. We optimized this circuit by replacing it with a circuit consisting of 48 internal wires and fifteen 3-to-1 gates and one 2-to-1 gate. We only looked at the case of P_2 obtaining the result, the extension to the first party obtaining the result is standard and requires an extension to the circuit to be made, for which similar optimizations can be made.

The size of the modified circuit: Step 0 of the protocol replaces the circuit with a different one which has $\max(4n, 8s_2)$ input wires. The statistical security parameter s_2 therefore affects the size of the circuit, both in terms of the number of wires and the number of gates. When $n < 2s_2$, as in our experiments, we have $8s_2$ new input wires. Each original input wire is replaced with the exclusive-or of about $4s_2$ input wires, which can be computed using $4s_2 - 1$ gates. The circuit therefore grows by about $4ns_2$ gates, which in our case translate to 2560 gates for $s_2 = 40$, and 3840 gates for $s_2 = 60$. We managed to optimize this construction by using a variant of structured Gaussian elimination in order to reuse gates. As a result, for the case of $s_2 = 40$, the augmented circuit produced in Stage 0 has over one thousand gates and over one thousand five hundred internal wires. If s_2 is increased to 60 then the augmented circuit now has over one thousand five hundred gates and over two thousand internal wires. The exact increase in size depends on the random choices made in Stage 0, but the above values are indicative.

Implementation: The program was implemented in C++ using standard libraries; the elliptic curve routines made use of specially written assembly functions to perform the arithmetic instructions. On the machine that was used for the experiments, and the curve we were using, the software needed 3.9 milliseconds for a basic multiplication, 1.2 milliseconds to multiply the fixed generator, and 5.1 milliseconds in order to compute $(aP + bQ)$ (using a variant of the method described in Algorithm 14.88 of [16]).

The input to the program was a circuit represented by a text file, each line of the text file represented a gate. For example the line

```
2 1 0 16 32 0100
```

represents a 2-to-1 gate which has input wires numbered 0 and 16 and produces the output wire 32. The value of the gate is given by the string which follows. The above example implements a two-bit "less than" gate, namely it will output a 1 on wire 32 only if $w_0 < w_{16}$, i.e. the value of wire 0 is zero and the value of wire 16 is one.

Experiments: We performed a set of experiments with different values of the statistical security parameters s_1 and s_2, and using both the ROM and standard model versions of the protocol. The run times, in seconds, are presented in Table 1, and are reported for each step of the protocol. Timings are performed using the standard Linux system timing facilities, and are as such only indicative. The wall time is measured using the standard *time* function and the system and user times are measured using the *getrusage* function. The wall time represents

the elapsed wall clock time in running the program, the user time represents the amount of time each party actually performed some computation, whereas the syst time represents the time spent by each party in system routines (for example transmitting data, or writing to disk, etc.). All timings were performed on an Intel Core 2 6420 running at 2.13 GHZ with 4096 KB of cache and 2 GB of RAM and are given in seconds.

Basic observations: The computation is not instantaneous but overall the run time is quite reasonable (the overall run time is about 2-3 minutes for a security parameter $s_1 = 160$). The run time is affected, of course, by the fact that 160 copies of the circuit are being used in the computation (compared to a protocol secure against semi-honest adversaries, which uses only a single copy of the circuit), and the fact that each circuit is much larger than its original form (in the experiment more than 1000 gates are added to the circuit in Step 0, where the original circuit consisted of less than 20 gates).

Oblivious transfers: It is a little surprising that Step 2, which includes the oblivious transfers, is not the main bottleneck of the protocol. This is true even though we implemented an OT protocol which is secure against malicious adversaries according to a full simulation definition.

Preprocessing: About half of the run time is consumed by Step 1, where P_1 prepares the circuits and the commitments. This step can be run offline, before the inputs are known, reducing the online run time by about 50%.

Scaling: Increasing s_1 by a factor of c_1 increases by a factor of c_1^2 the number of commitments generated by P_1 in Step 1, and increases the number of circuits by c_1. Increasing s_2 by a factor of c_2 increases the size of the modified part of the circuit (which is the bulk of the circuit in our experiments) by a factor of c_2, and therefore the total size of the circuits is increased by a factor of $c_1 c_2$. In the experiments, we increased both s_1 and s_2 by a factor of 1.5 (from 40 to 60, and from 160 to 240, respectively). We therefore expected the overhead to increase by a factor of 2.25. The actual measurements showed an increase by a factor slightly larger than 2.

We did not conduct experiments with circuits of different sizes. When all other parameters are fixed, we expect the run time to be linear in the size of the *modified circuit* (after the modifications done in Step 0). We can estimate the size of the modified circuit as follows: If P_2 has n input wires in the original circuit, then the modified circuit is expected to have about $\frac{n}{2}\max(4n, 8s_2)$ more gates. (Applying structured Gaussian elimination can enable us to reuse gates and minimize the size of the modified circuit.)

Performance in the ROM and in the standard model: What is interesting about the timings is that there is very little difference between the timings in the ROM and those in the standard model. In Step 1 the ROM version is more efficient simply due to the slightly more efficient encryption scheme used.[7]

[7] The KDF is invoked in the standard model protocol about twice as many times as in the ROM protocol (since the encryption function in the standard model calls

Given the large number of encryptions needed to produce the garbled circuit this translates into a small advantage for the ROM version compared to the standard-model implementation. For a similar reason one obtains a performance improvement in the ROM in Step 7 in which the circuit is evaluated by P_2. The decrease in performance of the ROM compared to the standard model in Step 3 we cannot explain, but it is likely to be caused by experimental error.

In viewing the timings it should be born in mind that the main place that the random oracle model is used is in the oblivious transfers in Step 2. At this point we use the ROM to reduce the round complexity of the two required zero-knowledge proofs (see Appendix A for details of this). However, these two proofs are only used once in the whole run of the protocol as we have batched the oblivious transfers, and therefore the run time of Step 2 is about the same in both the ROM and the standard model protocols.

What is surprising about the fact that the standard model is comparable in performance to the ROM is that for simpler cryptographic functionalities, such as encryption or signature schemes, the performance of the best ROM based scheme is often twice as fast as the best known standard model scheme.

6 Future Work

An obvious task is to develop the current implementation into a complete system for secure computation. In particular, the system should include a front end that will enable users to provide a high-level specification of the function that they want to compute, and specify the different security parameters that shall be used. A natural approach for this task would be to modify the FairPlay compiler [15] to support our implementation.

The performance of the system is greatly affected by the circuit modification in Step 0 of the protocol, which increases the number of inputs and the size of the circuit. We implemented this step according to the randomized construction in [13]. Another option is to use a linear error-correction code for defining the relation between the original and new input wires of the circuit. (A careful examination of the proof in [13] shows that this is sufficient.) We need an $[N, k, d]$ linear binary code which encodes k bit words into N bit words with a distance of $d = s_2$ (say, $d = 40$). The parameter k corresponds to the number of original input wires of P_2, while N corresponds to the number of new input wires. The code should satisfy two criteria: (1) the rate k/N should be as high as possible, to keep the number of new input wires close to the number of original input wires, and (2) the block length k should be minimized, to enable the code to be applied (and the rate k/N to be achieved) even if P_2's input is relatively short.

---

the KDF twice). The increase in the run time of Step 1 when changing the ROM implementation to the standard-model implementation (for $s_1 = 160$) is from 60sec to 67sec. We therefore estimate that the circuit construction (Step 1(a)) takes about 7 seconds in the ROM protocol and 14 seconds in the standard model protocol.

Run Times in the Random Oracle Model

Time	Step 1	2	3	4	5	6	7	8	Total
P_1, $s_1 = 160$, $s_2 = 40$									
Wall	74	20	24	0	7	10	0	0	135
User	60	17	12	0	3	4	0	0	
Syst	16	2	3	0	0	0	0	0	
P_2, $s_1 = 160$, $s_2 = 40$									
Wall	74	20	24	0	8	9	35	1	171
User	0	8	14	0	8	7	29	1	
Syst	0	0	10	0	2	4	8	0	
P_1, $s_1 = 240$, $s_2 = 60$									
Wall	159	34	51	0	19	13	0	0	276
User	123	30	24	0	11	6	0	0	
Syst	35	2	9	0	1	0	0	0	
P_2, $s_1 = 240$, $s_2 = 60$									
Wall	159	34	51	0	19	13	78	3	358
User	0	12	28	0	17	10	61	2	
Syst	0	0	22	0	7	5	18	0	

Run Times in the standard Model

Time	Step 1	2	3	4	5	6	7	8	Total
P_1, $s_1 = 160$, $s_2 = 40$									
Wall	84	20	24	0	7	7	0	0	142
User	67	18	10	0	5	3	0	0	
Syst	15	0	5	0	0	0	0	0	
P_2, $s_1 = 160$, $s_2 = 40$									
Wall	84	20	24	0	7	7	40	2	184
User	0	10	13	0	7	5	32	4	
Syst	0	0	11	0	1	3	8	2	
P_1, $s_1 = 240$, $s_2 = 60$									
Wall	181	35	45	0	18	12	0	0	291
User	145	30	24	0	8	8	0	0	
Syst	35	0	7	0	1	2	0	0	
P_2, $s_1 = 240$, $s_2 = 60$									
Wall	181	35	45	0	18	12	87	5	362
User	0	12	23	0	15	9	70	7	
Syst	0	0	21	0	4	3	20	0	

Table 1. Run times of our experiments.

References

1. B. Aiello, Y. Ishai, and O. Reingold. Priced Oblivious Transfer: How to Sell Digital Goods. In *EUROCRYPT 2001*, Springer-Verlag (LNCS 2045), 119–135, 2001.
2. G. Aggarwal, N. Mishra, and B. Pinkas. Secure Computation of the k-th Ranked Element. In *EUROCRYPT 2004*, Springer-Verlag (LNCS 3027), 40–55, 2004.
3. A. Ben-David, N. Nisan and B. Pinkas, FairplayMP – A System for Secure Multi-Party Computation, manuscript, 2008.
4. P. Bogetoft, D.L. Christensen, I. Dåmgard, M. Geisler, T. Jakobsen, M. Krøigaard, J.D. Nielsen, J.B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach and T. Toft, Multiparty Computation Goes Live, Cryptology ePrint Archive 2008/068, 2008.
5. P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter. and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography and Data Security – FC 2006*, Springer-Verlag LNCS 4107, 142–147, 2006.
6. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
7. D. Chaum and T.P. Pederson. Wallet Databases with Observers. In *Advances in Crytology – Crypto '92*, Springer-Verlag LNCS 740, 89–105, 1992.
8. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge Univ. Press, 2004.
9. O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

10. C. Hazay and Y. Lindell. Oblivious transfer, polynomial evaluation and set intersection. Manuscript, 2008.
11. S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *Advances in Cryptology – EuroCrypt 2007*, LNCS 4515, 97–114, 2007.
12. Y.T. Kalai. Smooth Projective Hashing and Two-Message Oblivious Transfer. In *EUROCRYPT 2005*, Springer-Verlag (LNCS 3494), pages 78–95, 2005.
13. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007*, Springer-Verlag LNCS 4515, 52–78, 2007.
14. D. Malkhi and M.K. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography – PKC 2006*, LNCS 3958, 458–473, 2006.
15. D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay — a secure two-party computation system. In *Proc. of 13th USENIX Security Symposium*, 2004.
16. A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.
17. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *12th SODA*, 448-457, 2001.
18. T.P. Pederson. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advanced in Cryptology – CRYPTO '91*, LNCS 576, 129–140, 1992.
19. *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography*. Available from `http://www.secg.org/download/aid-385/sec1_final.pdf`
20. SECG. *Standards for Efficient Cryptography, SEC 2: Recommended elliptic curve domain parameters*. Available from `http://www.secg.org`.
21. C.P. Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology – Crypto '89*, Springer-Verlag LNCS 435, 239-252, 1990.
22. V. Shoup. Sequences of games: A tool for taming complexity in security proofs. Manuscript, 2004.
23. D. Woodruff. Revisiting the Efficiency of Malicious Two-Party Computation. In *Eurocrypt '07*, Springer-Verlag (LNCS 4515), pages 79–96, 2007.
24. A. Yao. How to generate and exchange secrets. In *27th FOCS*, 162–167, 1986.

A Zero Knowledge Proofs

We now describe the zero-knowledge proof-of-knowledge protocols required in the OT protocol. In the ROM we use the standard Fiat-Shamir transform of an interactive honest-verifier Σ-protocol into a non-interactive protocol via hashing the commitment with the random oracle so as to produce the random challenge.

In the standard model we need to cope with non-honest verifiers by getting the verifier to commit to his challenge before the prover's commitment is issued. We use a highly-efficient transformation described in [10] to transform an honest-verifier Sigma protocol to a protocol that is a zero-knowledge proof of knowledge (the transformation is proven secure under the assumption that the discrete logarithm problem is hard and hence is highly suitable for proofs of Diffie-Hellman type statements).

A.1 $DL(Q; x)$

We assume a prover *Pro* who knows x and a verifier *Ver* who only knows Q

and P. The two protocols, one in the ROM and one in the standard model, are presented in Fig. 3. They are based on the HVZK proof of Schnorr [21].

Figure 3 ROM and non-ROM zero-knowledge proof of knowledge of discrete logarithms

ROM Version

- *Pro* computes $k \leftarrow \mathbb{Z}_q$, $R \leftarrow [k]P$, $s \leftarrow H(R)$, $z \leftarrow xs + k$. It sends R and z to *Ver*.
- *Ver* computes $s \leftarrow H(R)$. and accepts if $[z]P = [s]Q + R$.

Non-ROM Version

- *Pro* computes $a \leftarrow \mathbb{Z}_q$, $A \leftarrow [a]P$. It sends A to *Ver*.
- *Ver* computes $s, t \leftarrow \mathbb{Z}_q$, $C \leftarrow [s]P + [t]A$. and sends C to *Pro*.
- *Pro* computes $k \leftarrow \mathbb{Z}_q$, $R \leftarrow [k]P$. and sends R to *Ver*.
- *Ver* sends s, t to *Pro*.
- *Pro* checks whether $C = [s]P + [t]A$. and sends $z \leftarrow xs + k$ and a to *Ver*.
- *Ver* accepts if $[z]P = [s]Q + R$ and $A = [a]P$.

A.2 $DDH(P, [a]P, [b]P, [ab]P)$

We assume a prover *Pro* who knows b and a verifier *Ver* who only knows the four protocol inputs P, $Q = [a]P$, $U = [b]P$ and $V = [b]Q$. The two variants of the protocol are given in Fig. 4, both are based on the HVZK protocol from [7].

Figure 4 ROM and non-ROM zero-knowledge proof of knowledge of DDH tuple

ROM Version

- *Pro* computes $r \leftarrow \mathbb{Z}_q$, $A \leftarrow [r]P$, $B \leftarrow [r]Q$, $s \leftarrow H(A\|B)$, $z \leftarrow bs + r$. and sends A, B and z to *Ver*.
- *Ver* computes $s \leftarrow H(A\|B)$ and accepts if $[z]P = [s]U + A$ and $[z]Q = [s]V + B$.

Non-ROM Version

- *Pro* computes $w \leftarrow \mathbb{Z}_q$, $W \leftarrow [w]P$ and sends V to *Ver*.
- *Ver* computes $s, t \leftarrow \mathbb{Z}_q$, $C \leftarrow [s]P + [t]A$ and sends C to *Pro*.
- *Pro* computes $r \leftarrow \mathbb{Z}_q$, $A \leftarrow [r]P$, $B \leftarrow [r]Q$ and sends A and B to *Ver*.
- *Ver* sends s, t to *Pro*.
- *Pro* checks whether $C = [s]P + [t]V$. and sends $z \leftarrow bs + r$ and w to *Ver*.
- *Ver* accepts if $[z]P = [s]U + A$, $[z]Q = [s]V + B$ and $W = [w]P$.