The Simplest Protocol for Oblivious Transfer

Tung Chou[1] and Claudio Orlandi[2]

[1] Technische Universieit Eindhoven
[2] Aarhus University

Abstract Oblivious Transfer (OT) is the fundamental building block of cryptographic protocols. In this paper we describe the simplest and most efficient protocol for 1-out-of-2 OT to date, which is obtained by tweaking the Diffie-Hellman key-exchange protocol. The protocol achieves UC-security against active corruptions in the random oracle model. Due to its simplicity, the protocol is extremely efficient and it allows to perform n 1-out-of-m OTs using only:

- **Computation:** $(m+1)n+2$ exponentiations (mn for the receiver, $mn+2$ for the sender) and
- **Communication:** $32(n+1)$ bytes (for the group elements), and $2mn$ ciphertexts.

We also report on an implementation of the protocol using elliptic curves, and on a number of mechanisms we employ to ensure that our software is secure against active attacks too. Experimental results show that our protocol (thanks to both algorithmic and implementation optimizations) is at least one order of magnitude faster than previous work.

1 Introduction

Oblivious Transfer (OT) is a cryptographic primitive defined as follows: in its simplest flavour, 1 out of 2 OT, a sender has two input messages M_0 and M_1 and a receiver has a choice bit c. At the end of the protocol the receiver is supposed to learn the message M_c and nothing else, while the sender is supposed to learn nothing. Perhaps surprisingly, this extremely simple primitive is sufficient to implement any cryptographic task [Kil88]. OT is also necessary to implement most advanced cryptographic tasks, such as secure two-party computation (e.g., the millionaire's problem).

Given the importance of OT, and the fact that most OT applications require a very large number of OTs, it is crucial to construct OT protocols which are at the same time efficient and secure against realistic adversaries.

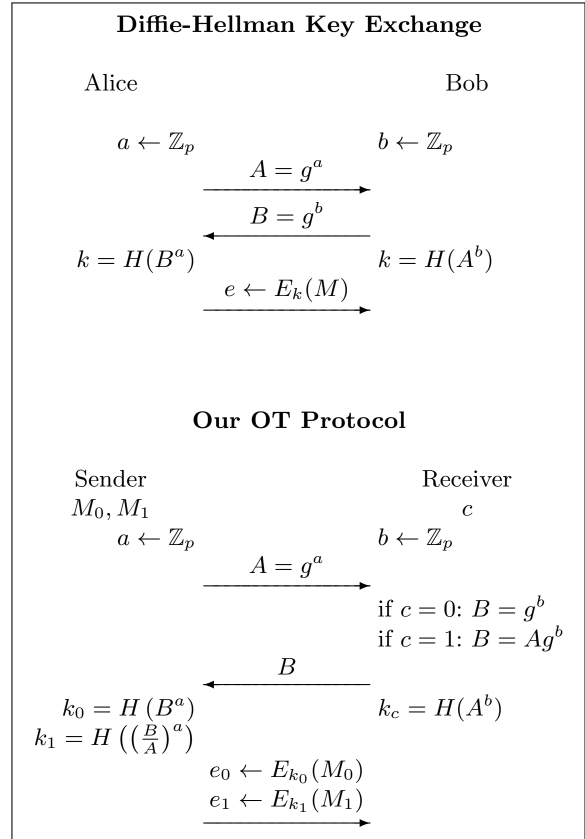

Figure 1. Our protocol in a nutshell

A Novel OT Protocol. In this paper we present an extremely *simple*, *efficient* and *secure* OT protocol which has (to the best of our knowledge) not appeared in the scientific literature before. The protocol is a simple tweak of the celebrated Diffie-Hellman (DH) key exchange protocol. Given a group $\mathbb{G}$ and a generator g, the DH protocol allows two players Alice and Bob to agree on a key as follows: Alice

samples a random a, computes $A = g^a$ and sends A to Bob. Symmetrically Bob samples a random b, computes $B = g^b$ and sends B to Alice. Now both parties can compute $g^{ab} = A^b = B^a$ from which they can derive a key k. The key observation is now that Alice can also derive a different key from the value $(B/A)^a = g^{ab-a^2}$, and that Bob cannot compute this group element (assuming that the computational DH problem is hard).

We can now turn this into a random OT protocol by letting Alice play the role of the sender and Bob the role of the receiver (with choice bit c) as shown in Figure 1. The first message (from Alice to Bob) is left unchanged (and can be reused over multiple instances of the protocol) but now Bob computes B as a function of his choice bit c: if $c = 0$ Bob computes $B = g^b$ and if $c = 1$ Bob computes $B = Ag^b$. At this point Alice derives two keys k_0, k_1 from $(B)^a$ and $(B/A)^a$ respectively. It is easy to check that Bob can derive the key k_c corresponding to his choice bit from A^b, but cannot compute the other one. (The protocol can be easily extended to 1-out-of-m OT.) Finally we prove that if we combine our novel *random OT* protocol with the right symmetric encryption scheme (e.g., an *authenticated encryption scheme*), then the overall protocol is secure in a strong, simulation based sense and in particular we achieve UC-security against active corruptions in the random oracle model.

A Secure and Efficient Implementation for the Random OT Protocol. We report on an efficient and secure implementation of the random OT protocol[3]: Our choice for the group is a twisted Edwards curve that has been used by Bernstein, Duif, Lange, Schwabe and Yang for building a *high-speed high-security* signature scheme [BDL+11]. The security of the curve comes from the fact that it is birationally equivalent to Bernstein's Montgomery curve Curve25519 [Ber06] where ECDLP is believed to be hard: Bernstein and Lange's SafeCurves website [BL14] reports cost of $2^{125.8}$ for solving ECDLP on Curve25519 using the *rho method*. The speed comes from the complete formulas for twisted Edwards curves proposed by Hisil, Wong, Carter, and Dawson in [HWCD08].

We first modify the code in [BDL+11] and build a fast implementation. In order to make use of the natural parallelism in the protocol, we also build a vectorized implementation that targets for the Intel Sandy Bridge and Ivy Bridge microarchitectures. A comparison with the state of the art shows that our implementation is at least an order of magnitude faster than previous work (we compare in particular with the implementation reported by Asharov, Lindell, Schneider and Zohner in [ALSZ13]). Furthermore, we take great care to make sure that our implementation is secure against both passive attacks (our software is *immune to timing attacks*, since the implementation is *constant-time*) and active attacks (by designing an appropriate encoding of group elements, which can be efficiently verified and computed on). Our code can be downloaded from `http://orlandi.dk/simpleOT`.

Organization. The rest of the paper is organized as follows: in Section 1.1 we discuss related work; in Section 2 we formally describe and analyse our protocol; Section 3 describes the chosen representation of group elements; Section 4 describes how group operations are performed; Section 5 describes the low level building blocks of the group operations; and Section 6 reports the timings of our implementation.

[3] Here *random* refers to the fact that at the end of the protocol the sender receives two random messages, which can be used later to encrypt his actual inputs. Our implementation does not include the encryption step, since random OT is enough for several applications.

1.1 Related Work

OT owes its name to Rabin [Rab81], but a similar concept was introduced few years earlier by Wiesner [Wie83] under the name of "conjugate coding". There are different flavours of OT, and in this paper we focus on the most common and useful flavour, namely 1-out-of-2 OT, which was first introduced in [EGL85]. Many efficient protocols for OT have been proposed over the years. Some of the protocols which are most similar to ours are those of Bellare-Micali [BM89] and Naor-Pinkas [NP01]. However, these protocols are still more complex than our and most importantly, are not not know to achieve full simulation based security. More recent OT protocols such as [HL10, DNO08, PVW08] focus on achieving a strong level of security in concurrent settings[4] without relying on the random oracle model. Unfortunately this makes these protocols more cumbersome for practical applications: even the most efficient of these protocols i.e., the protocol of Peikert, Vaikuntanathan, and Waters [PVW08] requires 11 exponentiations for each OT and a common random string (which must be generated by some trusted source of randomness at the beginning of the protocol and after the adversary chooses whether to corrupt the sender or the receiver). In comparison our protocol is more practical since it uses only $3n + 2$ exponentiations and does not require any (hard to implement in practice) setup assumptions.

OT Extension. While OT provably requires "public-key" type of assumptions [IR89] (such as factoring, discrete log, etc.), OT can be "extended" [Bea96] in the sense that it is enough to generate few "seed" OTs based on public-key cryptography which can then be extended to any number of OTs using symmetric-key primitives only (PRG, hash functions, etc.). This can be seen as the OT equivalent of *hybrid encryption* (where one encrypts a large amount of data using symmetric-key cryptography, and then encapsulates the symmetric-key using a public-key cryptosystem). OT extension can be performed very efficiently [IKNP03, ALSZ13] when one only wants security against passsive adversaries and relatively efficiently [Nie07, NNOB12, Lar14, ALSZ15] if one wants security against active adversaries. Still, to bootstrap OT extension we need a secure and efficient OT protocol for the seed OTs (as much as we need secure and efficient public-key encryption schemes to bootstrap hybrid encryption): The OT extension of [ALSZ15] reports that it takes time $(7 \cdot 10^5 + 1.3n)\mu s$ to perform n OTs, where the fixed term comes from running 190 base OTs. Using our protocol as the base OT in [ALSZ15] reduces the initial cost to approximately $190 \cdot 114 \approx\ 2 \cdot 10^4 \mu s$ [Sch15]., which leads to a significant overall improvement (e.g., a factor 10 for up to $4 \cdot 10^4$ OTs and a factor 2 for n up to $5 \cdot 10^5$ OTs).

2 The Protocol

We want to implement n $\binom{m}{1}$-OT for messages of length ℓ with κ-bit security between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$.[5] The sender $\mathcal{S}$ has m vectors of message $\{(M_0^i, \ldots, M_{m-1}^i)\}_{i\in\{1,\ldots,n\}}$ where for all $i, j : M_i^j \in \{0,1\}^\ell$. The receiver $\mathcal{R}$ has n choice values $c^i \in \{0, \ldots, m-1\}$ for $i = 1, \ldots, n$. At the end of the protocol $\mathcal{S}$ learns nothing and $\mathcal{R}$ learns $z^i = M_c^i \in \{0,1\}^\ell$ for all i's.

We split the presentation of the protocol in two parts: in the first part, we describe and analyze a protocol for *random OT* where the sender outputs m random keys and the receiver learns only one of them. Then we describe how to combine this protocol with an appropriate encryption scheme to achieve UC security.

[4] I.e., UC security [Can01], which is impossible to achieve without some kind of trusted setup assumptions [CF01].

[5] We describe the protocol performing n OTs in parallel since we can do this more efficiently than simply repeating n times the protocol for a single OT.

Notation. If S is a set $s \leftarrow S$ is a random element sampled from s. We work over an additive group $(\mathbb{G}, B, p, +)$ of prime order p (with $\log(p) > \kappa$) generated by B (the base point), and we use the additive notation for the group since we later implement our protocol using elliptic curves. Given the representation of some group element P we assume it is possible to efficiently verify if $P \in \mathbb{G}$.

Building Blocks. We use a cryptographic keyed hash-function (or a key-derivation function) $H : (\mathbb{G} \times \mathbb{G}) \times \mathbb{G} \to \{0,1\}^\kappa$ which is used to extract an κ bit key from a group element, and the first two inputs are used to seed the function.[6] We will model H as a random oracle when arguing about the security of our protocol.

2.1 Random OT

We are now ready to describe our *random OT* protocol:

Setup: (only once, independent of n):

1. $\mathcal{S}$ samples $y \leftarrow \mathbb{Z}_p$ and computes $S = yB$ and $T = yS$;
2. $\mathcal{S}$ sends S to $\mathcal{R}$, who aborts if $S \notin \mathbb{G}$;

Choose: (in parallel for all $i \in \{1, \ldots, n\}$)

1. $\mathcal{R}$ samples $x^i \leftarrow \mathbb{Z}_p$ and computes

$$R^i = c^i S + x^i B$$

2. $\mathcal{R}$ sends R^i to $\mathcal{S}$, who aborts if $R^i \notin \mathbb{G}$;

Key Derivation: (in parallel for all $i \in \{1, \ldots, n\}$)

1. For all $j \in \{0, \ldots, m-1\}$ $\mathcal{S}$ computes

$$k^i_j = H_{(S,R^i)}(yR^i - jT)$$

2. $\mathcal{R}$ computes

$$k^i_R = H_{(S,R^i)}(x^i S)$$

Basic Properties. It is easy to see that k^i_j is computed by hashing $x^i yB + (c^i - j)T$ and therefore at the end of the protocol $k^i_R = k^i_{c^i}$ if both parties are honest. It is also easy to see that:

Lemma 1. *No (computationally unbounded) $\mathcal{S}$ on input R^i can guess c^i w.p. greater than $1/p$.*

Proof. Since B generates $\mathbb{G}$, fixed any $P = x_0 B$ the probability that $R^i = P$ when $c^i = j$ is the probability that $x^i = (x_0 - c^i y)$, therefore $\forall S, P \in \mathbb{G}, j \in \{0, \ldots, m-1\}$, $\Pr[R^i = P | c^i = j] = 1/p$.

Lemma 2. *No (computationally bounded) $\mathcal{R}^*$ can output any two keys $k^i_{j_0}$ and $k^i_{j_1}$ with $j_0 \neq j_1 \in \{0, \ldots, m-1\}$ if the computational Diffie-Hellman problem is hard in $\mathbb{G}$.*

Proof. In the random oracle model $\mathcal{R}^*$ can only (except with negligible probability) compute $k^i_{j_0}, k^i_{j_1}$ by querying the oracle on points of the form $U^i_0 = (yR^i - j_0 T)$ and $U^i_1 = (yR^i - j_1 T)$. Assume for the sake of contradiction that there exist a PPT $\mathcal{R}^*$ who outputs $(R, j_0, j_1, U_0, U_1) \leftarrow \mathcal{R}^*(B, S)$ such that $(j_1 - j_0)^{-1}(U_0 - U_1) = T = \log_B(S)^2 B$ with probability at least ϵ. We show an algorithm $\mathcal{A}$ which on input $(B, X = xB, Y = xB)$ outputs $Z = xyB$ with probability greater than ϵ^3. Run

[6] Standard hash functions do not take group elements as inputs, and in later sections we will give explicit encodings of group elements into bitstrings.

$(R^X, U_0^X, U_1^X) \leftarrow \mathcal{R}^*(B, X)$, $(R^Y, U_0^Y, U_1^Y) \leftarrow \mathcal{R}^*(B, Y)$, then run $(R^+, U_0^+, U_1^+) \leftarrow \mathcal{R}^*(B, X+Y)$ and finally output

$$Z = \frac{(p+1)}{2}\left((U_0^+, U_1^+) - (U_0^X + U_1^X) - (U_0^Y + U_1^Y)\right)$$

Now $Z = xyB$ with probability at least ϵ^3, since when all three executions of $\mathcal{R}^*$ are successful, then $U_0^X + U_1^X = (x^2)B$, $U_0^Y + U_1^Y = (y^2)B$ and $U_0^+, U_1^+ = (x+y)^2B$ and therefore $Z = \frac{p+1}{2}2xyB = xyB$. $\square$

2.2 How to use the protocol and UC security

In this subsection we show that if we combine the random OT from the previous subsection with an appropriate encryption scheme, then the combined protocol achieves UC security.

Motivation. Lemma 1 and 2 only state that "privacy" holds for both the sender and the receiver. However, since OT is mostly used as a building block into more complex protocols, it is important to understand to which extent our protocol offers security when composed arbitrarily with itself or other protocols: Simulation based security is the minimal requirement which enables to argue that a given protocol is secure when composed with other protocols. Without simulation based security, it is not even possible to argue that a protocol is secure if it is executed twice in a sequential way! (See e.g., [DNO08] for a concrete counterexample for OT). The UC theorem [Can01] allows us to say that if a protocol satisfies the UC definition of security, then that protocol will be secure even when arbitrarily composed with other protocols. Among other things, to show that a protocol is UC secure one needs to show that a simulator can *extract* the input of a corrupted party: intuitively, this is a guarantee that the party *knows* its input, and its not reusing/modifying messages received in other protocols (aka malleability attack).

From Random OT to *standard* OT. We start by adding a transfer phase to the protocol, where the sender sends the encryption of his two messages to the receiver:

Transfer: (in parallel for all $i \in \{1, \ldots, n\}$)
1. For all $j \in \{0, \ldots, m-1\}$ $\mathcal{S}$ computes $e_j^i \leftarrow E(k_j^i, M_j^i)$
2. $\mathcal{S}$ sends $(e_0^i, \ldots, e_{m-1}^i)$ to $\mathcal{R}$;

Retrieve: (in parallel for all $i \in \{1, \ldots, n\}$)
1. $\mathcal{R}$ computes and outputs $z^i = D(k^i, e_{c^i}^i)$.

The encryption scheme. We need a symmetric encryption scheme (E, D). We call κ the bitlength of the key, ℓ the bitlength of the message and ℓ' the ciphertext bitlength. We allow the decryption algorithm to output a special symbol $\perp$ to indicate an invalid ciphertext. We need the encryption scheme to satisfy the following properties:

Definition 1. *We say a symmetric encryption scheme (E, D) is* non-committing *if there exist PPT algorithms $\mathcal{S}_1, \mathcal{S}_2$ such that $\forall m \in \{0,1\}^\ell$ (e', k') and (e, k) are computationally indistinguishable where $e' \leftarrow \mathcal{S}_1(1^\kappa)$, $k' \leftarrow \mathcal{S}_2(e', m)$, $k \leftarrow \{0,1\}^\kappa$ and $e \leftarrow E(k, m)$.*

The definition says that it is possible for a simulator to come up with a ciphertext e which can later be "explained" as an encryption of any message m, in such a way that the joint distribution of the encryption and the key in this simulated experiment is indistinguishable from the normal use of the encryption scheme, where a key is first sampled and then an encryption of m is generated.

Definition 2. (E, D) *satisfies* ciphertext integrity *if* $\Pr[D(k, e) \neq \perp | k \leftarrow \{0,1\}^\kappa, e \leftarrow \mathcal{A}(1^\kappa)]$ *is negligible in* κ.

Traditionally ciphertext integrity is defined for an adversary who has access to an encryption oracle, but the above definition suffices for our goal.

A concrete example. We give a concrete example of an encryption scheme satisfying Definition 1 and 2. In this encryption scheme $\kappa = \ell' = 2\ell$. The encryption algorithm $E(k, m)$ parses k as $(\alpha, \beta) \in GF(2^\ell) \times GF(2^\ell)$, outputs $c_1 = m + \alpha$ and $c_2 = \beta \cdot c_1$. The decryption function $D(k, e)$ parses k as $(\alpha, \beta) \in GF(2^\ell) \times GF(2^\ell)$, outputs $\perp$ if $c_2 \neq \beta \cdot c_1$ or $m = c_1 + \alpha$ otherwise. It is easy to see that this scheme satisfies Definition 1:[7] $\mathcal{S}_1$ outputs two random bitstrings $(c_1, c_2) \leftarrow GF(2^\ell) \times GF(2^\ell)$ and $\mathcal{S}_2(e, m)$ outputs $\alpha = c_1 + m$ and $\beta = c_2 \cdot c_1^{-1}$. Finally the scheme satisfies Definition 2 since (fixed any key k) only a fraction $2^{-\kappa}$ of ciphertexts do not decrypt to $\perp$.

Simulation Based Security (UC).[8] We can finally argue UC security of our protocol. In particular, we define a functionality $\mathcal{F}_{OT}^{-}(n, \ell)$ as follows: the functionality receives as input a vector of bits $(c^1, \ldots, c^n)$ from the receiver and a a vector of pairs of ℓ-bit messages $((M_0^1, M_1^1), \ldots, (M_0^1, M_1^1))$ from the sender, and outputs a vector of ℓ-bit strings $(z^1, \ldots, z^n)$ to the receiver, such that for all i $z^i = M_{c^i}^i$. In addition, we weaken the ideal functionality in the following way: a corrupted receiver can input the choice bits in an adaptive fashion i.e., the ideal adversary can input a choice bit (for any i) learn message z^i, then chooses which choice bit to input next and so on.

Theorem 1. *If the computational DH problem is hard in* $\mathbb{G}$ *the protocol above securely implements the functionality* $\mathcal{F}_{OT}^{-}(n, \ell)$ *in the random oracle model.*

The main idea behind the proof are: 1) it is possible to extract the choice value by checking whether a corrupted receiver queries the random oracle on points $yR^i + cT$ for some c (no adversary can query on points of this form for more than one c without breaking the CDH assumption and the *non-committing* property of (E, D) allows us to complete a successful simulation even if the corrupted receiver queries the oracle *after* he receives the ciphertexts) and 2) it is possible to extract the sender messages by decrypting the ciphertexts with every key which the receiver got from the random oracle (and the *ciphertext-integrity* property of (E, D) allows us to conclude that except with negligible probability D returns $\perp$ for all keys different from the correct one).

Proof. (Corrupted Sender) First we argue that our protocol securely implement the functionality against a corrupted sender in the random oracle model (we will in particular use the property that the simulator can learn on which points the oracle was queried on), by constructing a simulator for a corrupted $\mathcal{S}^*$ in the following way:[9] 1) in the first phase, the simulator answers random oracle queries $H_{(\cdot,\cdot)}(\cdot)$ at random; 2) at some point $\mathcal{S}^*$ outputs S and the simulator checks that $S \in \mathbb{G}$ or aborts otherwise; 3) the simulator now chooses a random x^i for all i and sends $R^i = x^i S$ to $\mathcal{S}^*$. Note that since x^i is chosen at random the probability that $\mathcal{S}^*$ had queried any oracle $H_{(S,R^i)}(\cdot)$ before is negligible. At this point, any time $\mathcal{S}^*$ makes a query of the form $H_{(S,R^i)}(P^q)$, the simulator stores its random answers in $k^{i,q}$; 4) Now $\mathcal{S}^*$ outputs $(e_0^i, \ldots, e_{m-1}^i)$ and the simulator computes for all i, j the value M_j^i in the following way: for all q compute $D(k^{i,q}, e_j^i)$ and set M_j^i to be the first

[7] In fact the scheme satisfies Definition 1 in a stronger, information theoretic sense.

[8] This paragraph assumes that the reader is familiar with standard security definitions and proofs for two-party computation protocols such as those presented in [HL10].

[9] The main goal of this argument is to show that a corrupted sender *knows* the message vectors.

such value which is $\neq \perp$ (if any), or $\perp$ otherwise; 5) finally it inputs all the vectors $(M_0^i, \ldots, M_{m-1}^i)$ to the ideal functionality. We now argue that no distinguisher can tell a real-world view apart from a simulated view. This follows from Lemma 1 (the distribution of R^i does not depend on c^i), and that the output of the honest receiver can only be different if there exist a pair (i, j) such that the adversary queried the random oracle on a point $P' \neq yR^i - jT$ and $m' = D(k', e_i^j) \neq \perp$, where $k' = H_{(S,R^i)}(P')$. In this case the simulator will input $M_j^i = m'$ to the ideal functionality which could cause the honest party in the ideal world to output a different value than it would in the real world (if $c^i = j$). But this happens only with negligible probability thanks to the property of the encryption scheme (Definition 2).

(Corrupted Receiver) We now construct a simulator for a corrupted receiver[10]: 1) In the first phase, the simulator answers random oracle queries $H_{(\cdot,\cdot)}(\cdot)$ truly at random; 2) at some point the simulator samples a random y and outputs $S = yB$. Afterwards it keeps answering oracle queries at random, but for each query of the form $k^q = H_{(S,P^q)}(Q^q)$ it saves the triple (k^q, P^q, Q^q) (since y is random the probability that any query of the form $H_{(S,\cdot)}(\cdot)$ was performed before is negligible); 3) at some point the simulator receives a vector of elements R^i and aborts if $\exists i : R^i \notin \mathbb{G}$; 4) the simulator now initializes all $c^i = \perp$; for each tuple q in memory such that for some i it holds that $P^q = R^i$ the simulator checks if $Q^q = y(R^i - dS)$ for some $d \in \{0, \ldots, m-1\}$. Now the simulator saves this value d in c^i if c^i had not been defined before or aborts otherwise. In other words, when the simulator finds a candidate choice value d for some i it checks if it had already found a choice value for that i (i.e., $c^i \neq \perp$) and if so it aborts and outputs `fail`, otherwise if it had not found a candidate choice bit for i before (i.e., $c^i = \perp$) and in this case it sets $c^i = d$; 5) When the adversary is done querying the random oracle, the simulator has to send all ciphertexts vectors $\{(e_0^i, \ldots, e_{m-1}^i)\}_{i \in [i]}$: $\forall i \in [n], j \in \{0, \ldots, m-1\}$ the simulator sets a) if $c^i = \perp : e_i^j = \mathcal{S}_1(1^\kappa)$ b) if $j \neq c^i : e_i^j = \mathcal{S}_1(1^\kappa)$ and c) if $j = c^i : e_i^j = E(k_{c^i}^i, z^i)$; 6) at this point the protocol is almost over but the simulator can still receive random oracle queries. As before, the simulator answers them at random except if the adversary queries on some point $H_{(S,R^i)}(Q^q)$ with $Q^q = y(R^i - dS)$. If this happens for any i such that $c^i \neq \perp$ the simulator aborts and outputs `fail`. Otherwise the simulator sets $c^i = d$, inputs c^i to the ideal functionality, receives z^i and programs the random oracle to output $k' \leftarrow \mathcal{S}_2(e_{c^i}^i, z^i)$.

Now to conclude our proof, we must argue that a simulated view is indistinguishable from the view of a corrupted party in an execution of the protocol. When the simulator does not output `fail` indistinguishability follows immediately from Definition 1. Finally the simulator only outputs `fail` if $\mathcal{R}^*$ queries the oracle on two points U_0, U_1 such that $U_1 - U_0$ is a small multiple of $y^2 B$, and as argued in Lemma 2 such an adversary can be used to break the computational DH assumption. $\square$

Security in Practice. Clearly, a proof that $a \rightarrow b$ only says that b is true when a is true, and since cryptographic security models (a) are not always a good approximation of the real world, we discuss some of these discrepancies here and therefore to which extent our protocol is secure (b): When instantiating our protocol we must replace the random oracle with a hash function: the proof crucially relies on the fact that the oracle is *local* to the protocol i.e., it only exists while the protocol is running. Clearly, there is no such a thing in the real world. We argue here that our choice of using the transcript of the protocol (S, R^i) as salt for the hash function ensures to best possible extent that the oracle is *local* to the protocol. Consider the following man-in-the middle attack, where an adversary $\mathcal{A}$ plays two copies of the protocol (for simplicity here $m = 2$), one as

[10] The main goal of this argument is to show that a corrupted receiver *knows* the choice value.

the sender with $\mathcal{R}$ and one as the receiver with $\mathcal{S}$. Here is how the attack works: 1) $\mathcal{A}$ receives S from $\mathcal{S}$ and forwards it to $\mathcal{R}$; 2) Then the adversary receives R from $\mathcal{R}$ and sends $R' = S - R$ to $\mathcal{S}$; 3) Finally $\mathcal{A}$ receives e_0, e_1 from $\mathcal{S}$ and sends $e'_0 = e_1$ and $e'_1 = e_0$ to $\mathcal{R}$. It is easy to see that if the same hash function was used to instantiate the random oracle in the two protocols, then the honest receiver would output $z = m_{1-c}$, which is clearly a breach of security (i.e., this attack could not be run if the OT protocols are replaced with OT functionalities). This motivates our choice of using the transcript of the protocol (S, R^i) to salt the hash function. Now, if an adversary changes any message between the sender and the receiver, the keys obtained by the honest parties will be completely independent and therefore the receiver outputs $\perp$ except with negligible probability. This motivates in practice the need for an encryption scheme (E, D) which satisfies ciphertext integrity, and therefore we recommend to use our random OT protocol only together with an *authenticated encryption* scheme. Clearly this does not satisfy the *non-committing* property, but we conjecture that this does not lead to any concrete vulnerabilities.

3 The Random OT Protocol in Practice

This section describes how the random OT protocol can be realized in practice. In particular, this section focuses on describing how group elements are represented as bitstrings, i.e., the *encodings*. In the abstract description of the random OT protocol, the sender and the receiver transmit and compute on "group elements", but clearly any implementation of the protocol transmits and computes on bitstrings. We describe how the encodings are designed to achieve efficiency (both for communication and computation) and security (particularly against a malicious party who might try to send malformed encodings).

The Group. The group $\mathbb{G}$ we choose for the protocol is a subset of $\bar{\mathbb{G}}$; $\bar{\mathbb{G}}$ is defined by the set of points on the twisted Edwards curve

$$\{(x, y) \in \mathbb{F}_{2^{255}-19} \times \mathbb{F}_{2^{255}-19} : -x^2 + y^2 = 1 + dx^2y^2\}$$

and the twisted Edwards addition law

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 + x_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

introduced by Bernstein, Birkner, Joye, Lange, and Peters in [BBJ+08]. The constant d and the generator B can be found in [BDL+11]. The two groups $\bar{\mathbb{G}}$ and $\mathbb{G}$ are isomorphic respectively to $\mathbb{Z}_p \times \mathbb{Z}_8$ and $\mathbb{Z}_p$ with $p = 2^{252} + 27742317777372353535851937790883648493$.

Encoding of Group Element. An *encoding* $\mathcal{E}$ for a group $\mathbb{G}_0$ is a way of representing group elements as fixed-length bitstrings. We write $\mathcal{E}(P)$ for a bitstring which represents $P \in \mathbb{G}_0$. Note that there can be multiple bitstrings that represent P; if there is only one bitstring for each group element, $\mathcal{E}$ is said to be *deterministic* ($\mathcal{E}$ is said to be *non-deterministic* otherwise[11]). Also note that some bitstrings (of the fixed length) might not represent any group element; we write $\mathcal{E}(\mathbb{G}_1)$ for the set of bitstrings which represent some element in $\mathbb{G}_1 \subseteq \mathbb{G}_0$. $\mathcal{E}$ is said to be *verifiable* if there exists an efficient algorithm that, given a bitstring as input, outputs whether it is in $\mathcal{E}(\mathbb{G}_0)$ or not.

The Encoding $\mathcal{E}_X$ for Group Operations. The non-deterministic encoding $\mathcal{E}_X$ for $\bar{\mathbb{G}}$, which is based on the *extended coordinates* in [HWCD08], represents each point using the tuple $(X : Y : Z :$

[11] We stress that non-deterministic in this context does not mean that the encoding involves any randomness.

T) with $XY = ZT$, representing $x = X/Z$ and $y = Y/Z$. We use $\mathcal{E}_X$ whenever we need to perform group operations since given $\mathcal{E}_X(P), \mathcal{E}_X(Q)$ where $P, Q \in \bar{\mathbb{G}}$, it is efficient to compute $\mathcal{E}_X(P+P)$, $\mathcal{E}_X(P+Q)$, and $\mathcal{E}_X(P-Q)$. In particular, given an integer scalar r it is efficient to compute $\mathcal{E}_X(rB)$, and given r and $\mathcal{E}_X(P)$ it is efficient to compute $\mathcal{E}_X(rP)$. See Sections 4 and 5 for details on $\mathcal{E}_X$.

The Encoding $\mathcal{E}_0$ and Related Encodings. The deterministic encoding $\mathcal{E}_0$ for $\bar{\mathbb{G}}$ represents each group element as a 256-bit bitstring: the natural 255-bit encoding of y followed by a sign bit which depends only on x. The way to recover the full value x is described in [BDL+11, Section 5], and group membership can be verified efficiently by checking whether $x^2(y^2-1) = dy^2+1$ holds; therefore $\mathcal{E}_0$ is verifiable. See [BDL+11] for more details of $\mathcal{E}_0$.

For the following discussions, we define deterministic encodings $\mathcal{E}_1$ and $\mathcal{E}_2$ for $\mathbb{G}$ as

$$\mathcal{E}_1(P) = \mathcal{E}_0(8P), \mathcal{E}_2(P) = \mathcal{E}_0(64P), P \in \mathbb{G}.$$

We also define non-deterministic encodings $\mathcal{E}^{(0)}$ and $\mathcal{E}^{(1)}$ for $\mathbb{G}$ as

$$\mathcal{E}^{(0)}(P) = \mathcal{E}_0(P+t), \mathcal{E}^{(1)}(P) = \mathcal{E}_0(8P+t'), P \in \mathbb{G},$$

where t, t' can be any 8-torsion point. Note that each element in $\mathbb{G}$ has exactly 8 representations under $\mathcal{E}^{(0)}$ and $\mathcal{E}^{(1)}$.

Point Compression/Decompression. It is efficient to convert from $\mathcal{E}_X(P)$ to $\mathcal{E}_0(P)$ and back; since $\mathcal{E}_0$ represents points as much shorter bitstrings, these operations are called *point compression* and *point decompression*, respectively. Roughly speaking, point compression outputs $y = Y/Z$ along with the sign bit of $x = X/Z$, and point decompression first recovers x and then outputs $X = x, Y = y, Z = 1, T = xy$. We always check for group membership during point decompression.

We use $\mathcal{E}_0$ for data transmission: the parties send bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$ and expect to receive bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$. This means a computed point encoded by $\mathcal{E}_X$ has to be compressed before it is sent, and a received bitstring has to be decompressed for subsequent group operations. Sending compressed points helps to reduce the communication complexity: the parties only need to transfer $32 + 32n$ bytes in total.

Secure Data Transmission. At the beginning of the protocol $\mathcal{S}$ computes and sends $\mathcal{E}_0(S)$. In the ideal case, $\mathcal{R}$ should receive a bitstring in $\mathcal{E}_0(\mathbb{G})$ which he interprets as $\mathcal{E}_0(S)$. However, an attacker (a corrupted $\mathcal{S}^*$ or a man-in-the-middle) can send $\mathcal{R}$ 1) a bitstring that is not in $\mathcal{E}_0(\bar{\mathbb{G}})$ or 2) a bitstring in $\mathcal{E}_0(\bar{\mathbb{G}} \setminus \mathbb{G})$. In the first case, $\mathcal{R}$ detects that the received bitstring is not valid during point decompression and ignores it. In the second case, $\mathcal{R}$ can check group membership by computing the pth multiple of the point, but a more efficient way is to use a new encoding $\mathcal{E}'$ such that each bitstrings in $\mathcal{E}_0(\bar{\mathbb{G}})$ represents a point in $\mathbb{G}$ under $\mathcal{E}'$. Therefore $\mathcal{R}$ considers the received bitstring as $\mathcal{E}^{(0)}(S) = \mathcal{E}_0(S+t)$, where t can be any 8-torsion point.

The encoding $\mathcal{E}^{(0)}$ (along with point decompression) makes sure that $\mathcal{R}$ receives bitstrings representing elements in $\mathbb{G}$. However, an attacker can derive c^i by exploiting the extra information given by a nonzero t: a naive $\mathcal{R}$ would compute and send $\mathcal{E}_0(c^i(S+t) + x^iB) = \mathcal{E}_0(c^it + R^i)$; now by testing whether the result is $\mathcal{E}_0(\mathbb{G})$ the attacker learns whether $c^i = 0$.

To get rid of the 8-torsion point, $\mathcal{R}$ can multiply received point by $8 \cdot (8^{-1} \bmod p)$, but a more efficient way is to just multiply by 8 and then operate on $\mathcal{E}_X(8S)$ and $\mathcal{E}_X(8x^iB)$ to obtain and send $\mathcal{E}_1(R^i) = \mathcal{E}_0(8R^i)$, i.e, the encoding switches to $\mathcal{E}_1$ for R^i. After this $\mathcal{S}$ works similarly as $\mathcal{R}$: to ensure that the received bitstring represents an element in $\mathbb{G}$, $\mathcal{S}$ interprets the bitstring as

$\mathcal{S}$		
Output	Input	Operations
S	y	$y \cdot B$
$\mathcal{E}^{(0)}(S)$	S	$\mathcal{C}(S)$
$8S$	S	$8 \cdot S$
$\mathcal{E}_1(S)$	$8S$	$\mathcal{C}(8S)$
$64T$	$8y, 8S$	$8 \cdot (y \cdot 8S)$
$64R^i$	$\mathcal{E}^{(1)}(R^i)$	$8 \cdot \mathcal{D}(\mathcal{E}^{(1)}(R^i)$
$\mathcal{E}_2(R^i)$	$64R^i$	$\mathcal{C}(64R^i)$
$64yR^i$	$y, 64R^i$	$y \cdot 64R^i$
$\mathcal{E}_2(yR^i)$	$64yR^i$	$\mathcal{C}(64yR^i)$
$64(yR^i - T)$	$64T, 64yR^i$	$64yR^i - 64T$
$\mathcal{E}_2(yR^i - T)$	$64(R^i - T)$	$\mathcal{C}(64(yR^i - T))$

$\mathcal{R}$		
Output	Input	Operations
$8S$	$\mathcal{E}^{(0)}(S)$	$8 \cdot \mathcal{D}(\mathcal{E}^{(0)}(S))$
$\mathcal{E}_1(S)$	$8S$	$\mathcal{C}(8S)$
$8x^iB$	$8x^i$	$8x^i \cdot B$
$8S + 8x^iB$	$8S, 8x^i$	$8S + 8x^i$
$\mathcal{E}^{(1)}(R^i)$	$8R^i$	$\mathcal{C}(8R^i)$
$\mathcal{E}_2(R^i)$	$8R^i$	$\mathcal{C}(8 \cdot 8R^i)$
$64x^iS$	$8x^i, 8S$	$8x^i \cdot 8S$
$\mathcal{E}_2(x^iS)$	$64x^iS$	$\mathcal{C}(64x^iS)$

Table 1. How the parties compute encodings of group elements: each row shows that the "Output" is computed given "Input" using the operations "Operations". The input might come from the output of a previous row, a received string (e.g., $\mathcal{E}^{(1)}(R^i)$), or a random scalar that the party generates (e.g., $8x^i$). The upper half of the table are the operations that does not depend on i, which means the operations are performed only once for the whole protocol. $\mathcal{E}_X$ is suppressed: group elements written without encoding are actually encoded by $\mathcal{E}_X$. $\mathcal{C}$ and $\mathcal{D}$ stand for point compression and point decompression respectively. Computation of the rth multiple of P is denoted as "$r \cdot P$". In particular, $8 \cdot P$ can be carried out with only 3 point doublings.

$\mathcal{E}^{(1)}(R^i) = \mathcal{E}_0(8R^i + t)$; to get rid of the 8-torsion point $\mathcal{S}$ also multiplies the received point by 8, and then $\mathcal{S}$ operates on $\mathcal{E}_X(64R^i)$ and $\mathcal{E}_X(64T)$ to obtain $\mathcal{E}_X(64(yR^i - jT))$.

Key Derivation. The protocol computes $H_{S,R^i}(P)$ where P can be x^iS, yR^i, or $yR^i - jT$ for $j \in \{0, \ldots, m-1\}$. This is implemented by hashing $\mathcal{E}_1(S) \,\|\, \mathcal{E}_2(R^i) \,\|\, \mathcal{E}_2(P)$ with Keccak [BDPVA09] with 256-bit output. The choice of encodings is natural: $\mathcal{S}$ computes $\mathcal{E}_X(S)$, and $\mathcal{R}$ computes $\mathcal{E}_X(8S)$; since multiplication by 8 is much cheaper than multiplication by $(8^{-1} \bmod p)$, we use $\mathcal{E}_1(S) = \mathcal{E}_0(8S)$ for hashing. For similar reasons we use $\mathcal{E}_2$ for R^i and P.

Actual Operations. For completeness, we present in Table 1 a full overview of operations performed during the protocol for the case of 1 out of 2 OT (i.e., $m = 2$).

4 Group Operations

This section describes how group operations in Section 3 are implemented, with focus on the most computationally intensive part of the protocol, namely the exponentiations.

Scalar Multiplications. Exponentiations on the curve $\bar{\mathbb{G}}$ are called *scalar multiplications*. More precisely, a scalar multiplication stands for computation of $\mathcal{E}_X(r \cdot P)$ where $P \in \bar{\mathbb{G}}$; the point P is called the *base point* for the scalar multiplication. We follow [BDL+11] to first compute the 253-bit integer $r \bmod p$ and then write it as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$, where $r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$.

There are mainly three types of scalar multiplications in the protocol; each of them accounts for n of the $3n + 2$ scalar multiplications (the constants 8 and 64 are suppressed):

- computation of $\mathcal{E}_X(y \cdot R^i)$ by $\mathcal{S}$,
- computation of $\mathcal{E}_X(x^i \cdot B)$ by $\mathcal{R}$,
- computation of $\mathcal{E}_X(x^i \cdot S)$ by $\mathcal{R}$.

Note that two more scalar multiplications are required for computing $\mathcal{E}_X(S)$ and $\mathcal{E}_X(T)$. We show below how each type is implemented in our software, using point additions/doublings and table lookups as building blocks.

Point Additions and Doublings. We use the formula in [HWCD08, Section 3.1] to compute the addition $(X_3 : Y_3 : Z_3 : T_3) = (X_1 : Y_1 : Z_1 : T_1) + (X_2 : Y_2 : Z_2 : T_2)$ with 9 field multiplications (including one multiplication by $2d$). For *precomputed* points we follow [BDL+11] to use $(X' : Y' : Z')$ representing $x = (Y' - X')/2$ and $y = (Y' + X')/2$ with $Z' = 2dxy$. The mixed addition $(X_3 : Y_3 : Z_3 : T_3) = (X_1 : Y_1 : Z_1 : T_1) + (X_2' : Y_2' : Z_2')$ then takes only 7 field multiplications. We also use the formula in [HWCD08, Section 3.3] to compute the doubling $(X_2 : Y_2 : Z_2 : T_2) = 2 \cdot (X_1 : Y_1 : Z_1 : T_1)$ with 4 field multiplications and 4 field squarings. See Section 5 for more details about field arithmetic. For the rest of the section points are written without encodings since it should be clear in the context whether they are precomputed points or not.

Table Lookups. Scalar multiplication algorithms described below compute many intermediate points before reaching the final results. Some intermediate points are "looked up" from a table instead of being computed with point additions/doublings. Side-channel-resistant table lookups requires arithmetic operations, and building the table itself also takes some computation (although sometimes it can be considered as precomputation). However, with reasonable parameter choices (e.g., size of table), the benefit it brings can be worthy the cost.

More precisely, our scalar multiplication algorithms look up the r_i-th multiple of a point P from a table, where $r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$. We follow [BDL+11] to use a table containing $P, 2P, 3P, 4P, 5P, 6P, 7P, 8P$ to achieve efficient table lookups. See Section 5 for more details.

Computing yR^i's. We use the *fixed-window method* for each i, which is the standard method for *variable-base* scalar multiplications: first compute $R^i, 2R^i, 3R^i, 4R^i, 5R^i, 6R^i, 7R^i, 8R^i$ with 4 point doublings and 3 point additions and store them in a table; then starting with looking up $P_{63} = y_{63}R^i$ in the table, keep computing $P_{j-1} = 16P_j + y_{j-1}R^i$ with 4 point doublings, 1 point addition, and 1 table lookup, until $P_0 = yR^i$. For n scalar multiplications this method takes $256n$ doublings, $66n$ additions, and $64n$ table lookups.

Computing x^iB's. These are *fixed-base* scalar multiplications; a straightforward method is as follows: for each i we obtain $16^j x_j^i B$ from a precomputed table with 1 table lookup, and then add all the results to obtain $x^iB = \sum_j 16^j x_j^i B$. For n scalar multiplications this method takes $63n$ mixed additions and $64n$ table lookups.

Computing x^iS's. This is similar to fixed-base scalar multiplications since the base point is the same. The algorithm is as follows: fix a parameter α which divides 64; starting with computing

$$P_{\alpha-1} = \sum_{j=0}^{64/\alpha-1} 16^{\alpha j}(x_{j\alpha+\alpha-1}^i S)$$

with $64/\alpha - 1$ additions and $64/\alpha$ table lookups, keep computing

$$P_{k-1} = 16P_k + \sum_{j=0}^{64/\alpha-1} 16^{\alpha j} x_{j\alpha+i-1}^i S,$$

with 4 doublings, $64/\alpha$ additions and $64/\alpha$ table lookups, until $P_0 = x^i S$. We need to generate a table for looking up all $16^{\alpha j}(x^i_{j\alpha+i-1}S)$ in advance, which takes $3+4(64-\alpha)+64/\alpha$ doublings and $192/\alpha$ additions. For n scalar multiplications, this method takes $3+4(64-\alpha)+64/\alpha+4(\alpha-1)n$ doublings, $192/\alpha+63n$ additions, and $64n$ table lookups. Note that when $\alpha = 64$ the algorithm becomes the fixed-window method. When $\alpha = 1$ the algorithm is similar to the algorithm for fixed-base scalar multiplications.

The point of this method is to prepare some multiples of S that can be used for all the scalar multiplications. The more scalar multiplications there are, the more precomputation (the smaller α) it is worth. The optimal value of α which minimizes the computation can be estimated given n and the cost for point addition, point doubling, and table lookup.

Relative Cost of the Scalar Multiplications. Assuming $\mathcal{S}$ and $\mathcal{R}$ has the same CPU speed. Among the three types of scalar multiplications, the computation of yR^i's is the most expensive for it cannot benefit from using the same base point. On the contrary, the computation of x^iB's is much cheaper because of the precomputed table: in our implementation it is around 3 times faster than computation of yR^i's. The cost for x^iS's is somewhere in between: when n is small (using bigger α) the cost is close to that of yR^i's; when n is big enough (using smaller α) the cost is close to that of x^iB's. Consequently, when n is small the latency of the protocol is dominated by $\mathcal{R}$; when n is big enough it is dominated by $\mathcal{S}$.

Choosing the Radix. While we use radix 16 for the scalars, one might think using some other radix such as 32 or 8 can be better. Since computation of yR^i's is the most expensive, let's consider whether switching to another radix would help in this case. Switching to another radix only makes small difference in number of doublings, so only additions and table lookups have to be considered. The number of additions and table lookups are roughly the same for any reasonable radix. In our implementation a point addition is roughly 2.85 times slower than a table lookup. Switching to radix 32 would decrease the number of additions and table lookups by 20% while roughly doubling cost for each table lookup; therefore switching to radix 32 does not seem to help. Similarly, switching to radix 8 also does not seem to help, so radix 16 seems to be the best choice for our implementation.

5 Field Arithmetic and Table Lookups

This section describes our implementation strategy for arithmetic operations in $\mathbb{F}_{2^{255}-19}$ and table lookups, which serve as low-level building blocks for the scalar multiplication algorithms in Section 4. Field operations are decomposed into double-precision floating-point operations using our strategy. A straightforward way for implementation is then using double-precision floating-point instructions. However, a better way to utilize the $64 \times 64 \to 128$-bit general multiplier is to decompose field operations into integer instructions as [BDL+11] does. The real reason we decide to use floating-point operations is that it allows us to use 256-bit vector instructions on the target microarchitectures, which are functionally equivalent to 4 double-precision floating-point instructions. The technique, which is called *vectorization*, makes our vectorized implementation run much faster than our non-vectorized implementation based on [BDL+11].

Representation of Field Elements. Each field element $x \in \mathbb{F}_{2^{255}-19}$ is represented as 12 *limbs* $(x_0, x_1, \ldots, x_{11})$ such that $x = \sum x_i$ and $x_i/2^{\lceil 21.25i \rceil} \in \mathbb{Z}$. Each x_i is stored as a double-precision floating-point number. Field operations are then carried out by limb operations such as floating-point additions and multiplications.

When a field element gets initialized (e.g., when obtained from a table lookup), each x_i uses no more than 21 bits of the 53-bit mantissa. However, after a series of limb operations, the number of bits x_i takes can grow. It is thus necessary to reduce the number of bits (in the mantissa) with carries before any precision is lost; see below for more discussions.

Field Arithmetic. Additions and subtractions of field elements are implemented in a straightforward way: simply adding/subtracting the corresponding limbs. This does increase the number of bits in the mantissa, but in our application it suffices to reduce bits only at the end of the multiplication function.

A field multiplication is divided into two steps. The first step is a schoolbook multiplication on the $2 \cdot 12$ input limbs, with reduction modulo $2^{255} - 19$ to bring the result back to 12 limbs. The schoolbook multiplication takes 132 floating-point additions, 144 floating-point multiplications, and a few more multiplications by constants to handle the reduction.

Let $(c_0, c_1, \ldots, c_{11})$ be the result after schoolbook multiplication. The second step is to perform carries to reduce number of bits in c_i. Carry from c_i to c_{i+1} (indices work modulo 12), which we denote as $c_i \to c_{i+1}$, is performed with 4 floating-point operations: $c \leftarrow c_i + \alpha_i$; $c \leftarrow c - \alpha_i$; $c_i \leftarrow c_i - c$; $c_{i+1} \leftarrow c_{i+1} + c$. The idea is to use $\alpha_i = 3 \cdot 2^{k_i}$ where k_i is big enough so that the less significant part of c_i are discarded in $c_i + \alpha_i$, forcing c to contain only the more significant part of c_i. For $i = 11$, one extra multiplication is required to scale c by $19 \cdot 2^{-255}$ before it is added to c_0.

A straightforward way to reduce number of bits in all limbs is to use the carry chain $c_0 \to c_1 \to c_2 \to \cdots \to c_{11} \to c_0 \to c_1$. The problem with the straightforward carry chain is that there is not enough instruction level parallelism to hide the 3-cycle latencies (see discussion below). To hide the latencies we thus interleave the following 3 carry chains:

$$\begin{aligned}
&c_0 \to c_1 \to c_2 \to c_3 \to c_4 \to c_5, \\
&c_4 \to c_5 \to c_6 \to c_7 \to c_8 \to c_9, \\
&c_8 \to c_9 \to c_{10} \to c_{11} \to c_0 \to c_1.
\end{aligned}$$

In total the multiplication function takes 192 floating-point additions/subtractions and 156 floating-point multiplications. When the input operands are the same, many limb products will repeat in the schoolbook multiplication; a field squaring is therefore cheaper than a field multiplication. In total the squaring function takes 126 floating-point additions/subtractions and 101 floating-point multiplications.

Table Lookups. The task here is to "look up" $r_i P$ where $r_i \in \{-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7\}$ from a table containing $P, 2P, 3P, 4P, 5P, 6P, 7P, 8P$. We follow [BDL+11] to look up $r_i P$ in 2 steps: load $|r_i|P$ from the table and then negate the point if r_i is negative.

In order to obtain one of the limb of one of 4 coordinates X, Y, Z, T of $|r_i|P$, we first initialize a limb v to the corresponding limb of $0P$ if $|r_i| = 0$; v is set to 0 otherwise. Then for each of the 8 candidate limbs w in the table, we mask w with one AND operation and then OR the result into v. Loading $|r_i P|$ thus takes $4 \cdot 12 \cdot (8 \cdot 2) = 768$ 64-bit logical operations, and a few more for initialization and computing the masks.

Negating a point in the extended coordinates is simply negating the X and T coordinates. Negating a field element can be easily done by negating each limb of it. Since each limb is a floating-point number, we negate each limb by XORing -0.0 into the limb; conditional negation is done by masking -0.0 before the XOR. A conditional negation thus takes $2 \cdot 12 = 24$ XORs and a

instruction	latency	throughput	description
`vandpd`	1	1	bitwise and
`vorpd`	1	1	bitwise or
`vxorpd`	1	1 (4)	bitwise xor
`vaddpd`	3	1	4-way parallel double-precision floating-point additions
`vsubpd`	3	1	4-way parallel double-precision floating-point subtractions
`vmulpd`	5	1	4-way parallel double-precision floating-point multiplications

Table 2. 256-bit vector instructions used in our implementation. Note that `vxorpd` has throughput of 4 when it has only one source operand.

few more operations for masking. In total our table lookup function for extended coordinates takes 848 64-bit logical operations.

When P is a precomputed point we use the 3-coordinate system instead of the extended coordinates. Using this representation reduces cost for loading $|r_i P|$, but the conditional negation takes more operations since it involves conditionally negating one coordinate and conditionally swapping the other two coordinates. In total our table lookup function for precomputed points takes 680 64-bit logical operations.

Vectorization. We decompose field arithmetic and table lookups into 64-bit floating-point and logical operations. The Intel Sandy Bridge and Ivy Bridge microarchitectures, as well as many recent microarchitectures, offer instructions that operate on 256-bit registers. Some of these instructions treat the registers as vectors of 4 double-precision floating-point numbers and perform 4 floating-point operations in parallel; there are also 256-bit logical instructions that can be viewed as 4 64-bit logical instructions. We thus use these instructions to run 4 scalar multiplications in parallel. Table 2 shows the instructions we use, along with their latencies and throughputs on the Sandy Bridge and Ivy Bridge given in Fog's well-known survey [Fog14].

6 Implementation Results

This section compares the speed of our implementation of 1 out of 2 OT (i.e., $m = 2$) with other similar implementations. Since our protocol is quite similar to the Diffie-Hellman key exchange protocol, we first compare our DH speeds with existing Curve25519 implementations. The experiments are carried out on two machines on the eBACS site for publicly verifiable benchmarks [BL15]: `h6sandy` (Sandy Bridge) and `h9ivy` (Ivy Bridge). Since our protocol can serve as the base OTs for an OT extension protocol, we also compare our speed with a base OT implementation presented in [ALSZ13], which is included in the Scapi multi-party computation library; the experiments are made on an Intel Core i7-3537U processor (Ivy Bridge) where each party runs on one core.

Comparing with Curve25519 Implementations. Table 3 compares our work with existing Curve25519 implementations. "Cycles to generate a public key" indicates the time to generate the public key given a secret key; the existing implementation we choose is an implementation by Andrew Moon [MF15]. "Cycles to compute a shared secret" indicates the time to generate the shared secret, given a secret key and a public key; the numbers for existing implementation are from the eBACS site. Note that since our software uses 4-way vectorization, the numbers in the table are the time for generating 4 public keys or 4 shared secrets divided by 4. All our timings are better than the timings of existing Curve25519 implementations.

	Machines	`h6sandy`	`h9ivy`
existing Curve25519 implementations	Cycles to compute a public key	89500	83636
	Cycles to compute a shared secret	194036	182708
this work	Cycles to generate a public key	61458	60853
	Cycles to compute a shared secret	182169	180343

Table 3. DH speeds of our work and existing Curve25519 implementations.

	n	4	8	16	32	64	128	256	512	1024
this work	Running time of $\mathcal{S}$	548	381	321	279	265	257	246	237	228
	Running time of $\mathcal{R}$	472	366	279	229	205	200	193	184	177
[ALSZ13]	Running time of $\mathcal{S}$	17976	10235	6132	4358	3348	2877	2650	2528	2473
	Running time of $\mathcal{R}$	16968	9261	5188	3415	3382	2909	2656	2541	2462

Table 4. Timings for per OT in kilocycles. Multiplying the number of kilocycles by 0.5 one can obtain the running time (in μs) on our test architecture.

Comparing with Scapi. Table 4 shows the timings of our implementation for the random OT protocol, along with the timings of a base-OT implementation presented in [ALSZ13]. The paper presents several base-OT implementations; the one we compare with is Miracl-based with "long term security" using random oracle (cf. [ALSZ13, Section 6.1]). The implementation uses the NIST K-283 curve and SHA-1 for hashing, and it is not a constant-time implementation. It turns out that our work is an order of magnitude faster for $n \in \{4, 8, \ldots, 1024\}$.

Acknowledgments. We are very grateful to: Daniel J. Bernstein and Tanja Lange for invaluable comments and suggestions regarding elliptic curve cryptography and for editorial feedback on earlier versions of this paper; Yehuda Lindell for useful comments on our proof of security; Peter Schwabe for various helps on implementation, including providing low-level code for field arithmetic.

Tung Chou is supported by Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005. Claudio Orlandi is supported by: the Danish National Research Foundation and The National Science Foundation of China (grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation; the Center for Research in Foundations of Electronic Markets (CFEM); the European Union Seventh Framework Programme ([FP7/2007-2013]) under grant agreement number ICT-609611 (PRACTICE).

References

ALSZ13. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer communications security*, pages 535–548. ACM, 2013.

ALSZ15. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. Cryptology ePrint Archive, Report 2015/061, 2015. `http://eprint.iacr.org/`.

BBJ+08. Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In *Progress in Cryptology–AFRICACRYPT 2008*, pages 389–405. Springer, 2008.

BDL+11. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer-Verlag Berlin Heidelberg, 2011.

BDPVA09. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3:30, 2009.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 479–488, 1996.

Ber06. Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

BL14. Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography, accessed 1 December 2014. `http://safecurves.cr.yp.to`.

BL15. Daniel J Bernstein and Tanja Lange. eBACS: Ecrypt benchmarking of cryptographic systems, accessed 16 March 2015. `http://bench.cr.yp.to`.

BM89. Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and spplications. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 547–557, 1989.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.

CF01. Ran Canetti and Marc Fischlin. Universally composable commitments. *IACR Cryptology ePrint Archive*, 2001:55, 2001.

DNO08. Ivan Damgård, Jesper Buus Nielsen, and Claudio Orlandi. Essentially optimal universally composable oblivious transfer. In *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*, pages 318–335, 2008.

EGL85. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

Fog14. Agner Fog. Instruction tables. 2014. `http://www.agner.org/optimize/instruction_tables.pdf`.

HL10. Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010.

HWCD08. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *Advances in Cryptology-ASIACRYPT 2008*, pages 326–343. Springer, 2008.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 145–161, 2003.

IR89. Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 44–61, 1989.

Kil88. Joe Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 20–31, 1988.

Lar14. Enrique Larraia. Extending oblivious transfer efficiently, or - how to get active security with constant cryptographic overhead. *IACR Cryptology ePrint Archive*, 2014:692, 2014.

MF15. Andrew Moon "Floodyberry". Implementations of a fast elliptic-curve digital signature algorithm, accessed 16 March 2015. `https://github.com/floodyberry/ed25519-donna`.

Nie07. Jesper Buus Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. Cryptology ePrint Archive, Report 2007/215, 2007. `http://eprint.iacr.org/`.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 681–700, 2012.

NP01. Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 448–457, 2001.

PVW08. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 554–571, 2008.

Rab81. Michael O. Rabin. How to exchange secrets with oblivious transfer. *Technical Report TR-81, Aiken Computation Lab, Harvard University*, 1981.

Sch15. Thomas Schneider. Personal communication, 2015.

Wie83. Stephen Wiesner. Conjugate coding. *SIGACT News*, 15(1):78–88, January 1983.