The Simplest Protocol for Oblivious Transfer

Tung Chou and Claudio Orlandi

Technische Universieit Eindhoven and Aarhus University

Abstract Oblivious Transfer (OT) is one of the fundamental building blocks of cryptographic protocols. In this paper we describe the simplest and most efficient protocol for 1-out-of-n OT to date, which is obtained by tweaking the Diffie-Hellman key-exchange protocol. The protocol allows to perform m 1-out-of-n OTs using only $2+3m$ full exponentiations ($2m$ for the receiver, $2+m$ for the sender) and, sending only $m+1$ 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.

Update: The proceeding version of this paper contains incorrect claims. See Section 1.1 for details.

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 can also be used to implement most advanced cryptographic tasks, such as secure two- and multi-party computation (e.g., the millionaire's problem) in an efficient way [NNOB12, BLN+15].

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.

A Novel OT Protocol. In this paper we present a novel and extremely *simple*, *efficient* and *secure* OT protocol. 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).

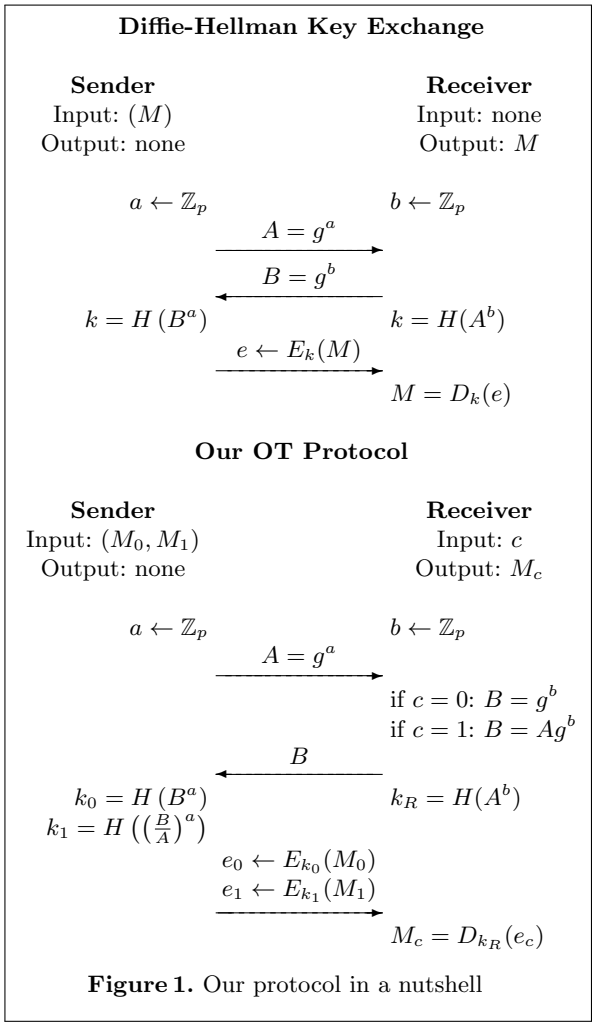

Figure 1. Our protocol in a nutshell

We can now turn this into an 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. This can be seen as a *random OT* i.e., an OT where the sender has no input but instead receives two random messages from the protocol, which can be used later to encrypt his inputs, thus achieving the OT functionality.

A Secure and Efficient Implementation. We report on an efficient and secure implementation of the 1-out-of-2 random OT protocol: Our choice for the group is a twisted Edwards curve that has been used by Bernstein, Duif, Lange, Schwabe and Yang for building the Ed25519 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 for a single OT. Later we build a vectorized implementation that runs OTs in batches. A comparison with the state of the art shows that our vectorized 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]) on recent Intel microarchitectures. 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 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 the low level building blocks of the group operations; and Section 5 reports the timings of our implementation.

Related Work. OT owes its name to Rabin [Rab81] (a similar concept was introduced 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 $\binom{n}{1}$-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]. More recent OT protocols such as [HL10, DNO08, PVW08] focus on achieving a strong level of security in concurrent settings[1] 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 a single $\binom{2}{1}$-OT and a common random string (which must be generated by some trusted source of randomness at the beginning of the protocol).

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 both against passive [IKNP03,ALSZ13] and active [Nie07,NNOB12,Lar14,ALSZ15,KOS15] 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 $(7 \cdot 10^5 + 1.3m)\mu s$ to perform m OTs, where the fixed term comes from running 190 base OTs. Using our protocol as the base OT in [ALSZ15] would reduce the initial cost to approximately

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

$190 \cdot 114 \approx 2 \cdot 10^4 \mu s$ [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 up to $5 \cdot 10^5$ OTs).

1.1 Incorrect Security Claim in Proceeding Version

The proceeding version of this work [CO15] claims that our protocol achieves UC security. The claim is incorrect, and has therefore been removed from this version.

Li and Micciancio [LM18] showed that the protocol cannot be simulated in the equational framework, due to subtle timing attacks. Genç, Iovino and Rial [GIR17] pointed out a problem with the proof of security, noticing that the protocol cannot be proven secure under the CDH assumption.[2] This particular problem was later fixed by Hauck and Loss using the GapDH assumption [HL17], who also proposed a different protocol based on the CDH assumption only.

It was later pointed out by several authors (Byali, Patra, Ravi and Sarkar [BPRS17], Doerner, Kondi, Lee and shelat [DKLs18]), that the extraction strategy in the case of a corrupt receiver in the original proof of security is incompatible with composable security. In a nutshell, the issue is that a corrupt sender can "attack" the protocol by delaying decryption. Thus, the simulator cannot extract the input of the receiver before the protocol is over (and cannot use said inputs to simulate later protocol messages e.g., when combining our OT with garbled circuits). It appears that this problem can be circumvented if the next protocol message is from the receiver to the sender, and this message is a "proof of timely decryption" in the sense that the sender will check this "proof" and only accept if indeed the the receiver has performed the necessary decryption queries. This technique has been employed by Barreto, David, Dowsley, Morozov and Nascimento in [BDD+17], where OT protocols in the random oracle from different assumptions are presented (their protocol contains an ad-hoc "proof of timely decryption") and in [DKLs18] (where our OT is combined with an OT extension protocol, which informally works as "proof of timely decryption"). See [CJS14] for a more thorough discussion of the issues arising in using random oracles in UC-proof of security, and for an OT protocol that can be proven secure in the "global" random oracle model.

2 The Protocol

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 $P \in \mathbb{G}$. We use $[n]$ as a shortcut for $\{0, 1, \ldots, n-1\}$.

Building Blocks. We use a hash-function $H : (\mathbb{G} \times \mathbb{G}) \times \mathbb{G} \to \{0,1\}^\kappa$ as a key-derivation function to extract a κ bit key from a group element, and the first two inputs are used to seed the function.[3] We model H as a random oracle when arguing about the security of our protocol.

Input/Outputs. We want to implement m $\binom{n}{1}$-OT's for ℓ-bit messages with κ-bit security between a sender $\mathcal{S}$ and a receiver $\mathcal{R}$. The receiver $\mathcal{R}$ has a vector of indices $(c^1, \ldots, c^m) \in [n]^m$, and the sender $\mathcal{S}$ has m vectors of message $\{(M_0^i, \ldots, M_{n-1}^i)\}_{i \in [m]}$ for all $i, j : M_i^j \in \{0,1\}^\ell$. At the end of the protocol the receiver $\mathcal{R}$ outputs a vector of ℓ-bit strings $(z^1, \ldots, z^n)$, such that for all $i \in [m]$, $z^i = M_{c^i}^i$.

2.1 Random OT

We split the presentation in two parts: first, we describe and analyze a protocol for *random OT* where the sender outputs n random keys and the receiver only learns one of them; then, we describe how to combine

[2] See also `https://eprint.iacr.org/forum/read.php?18,962`.

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

this protocol with an appropriate encryption scheme to complete the OT. We are now ready to describe our novel *random OT* protocol:

Setup: (only once, independent of m):
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 $i \in [m]$)
1. $\mathcal{R}$ (with input $c^i \in [n]$) 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 $i \in [m]$)
1. For all $j \in [n]$, $\mathcal{S}$ computes

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

2. $\mathcal{R}$ computes

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

Basic Properties. The key k_j^i is computed by hashing $x^i yB + (c^i - j)T$ and therefore at the end of the protocol $k_R^i = k_{c^i}^i$ 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 with probability greater than $1/n$.*

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 $\forall S, P \in \mathbb{G}, j \in [n]$, $\Pr[R^i = P | c^i = j] = 1/p$, which is independent of j.

Lemma 2. *No (computationally bounded) $\mathcal{R}^*$ can output any two keys $k_{j_0}^i$ and $k_{j_1}^i$ with $j_0 \neq j_1 \in [n]$ 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 $k_{j_0}^i, k_{j_1}^i$ by querying the oracle on points of the form $U_0^i = (yR^i - j_0 T)$ and $U_1^i = (yR^i - j_1 T)$. 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 $(B, X = xB, Y = yB)$ outputs $Z = xyB$ with probability greater than ϵ^3. Run $(R^X, U_0^X, U_1^X) \leftarrow \mathcal{R}^*(B, X)$, $(R^Y, U_0^Y, U_1^Y) \leftarrow \mathcal{R}^*(B, Y)$, 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 $U_0^+, U_1^+ = (x+y)^2 B$ and therefore $Z = \frac{p+1}{2} 2xyB = xyB$. □

Note that the above proof loses a cubic factor. A better proof for this lemma, which only loses a quadratic factor, can be found in [BCP04].

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

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

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

The protocol uses a symmetric encryption scheme (E, D). We call $\mathcal{K}, \mathcal{M}, \mathcal{C}$ the key space, message space and ciphertext space respectively and κ the security parameter. We allow the decryption algorithm to output a special symbol $\perp$ to indicate an invalid ciphertext. We want to use an encryption scheme that satisfies 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 $\forall M \in \mathcal{M}$ (e', k') and (e, k) are computationally indistinguishable where $e' \leftarrow \mathcal{S}_1(1^\kappa)$, $k' \leftarrow \mathcal{S}_2(e', M)$, $k \leftarrow \mathcal{K}$ and $e \leftarrow E(k, M)$ ($\mathcal{S}_1, \mathcal{S}_2$ are allowed to share a state).*

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, in such a way that the joint distribution of the ciphertext 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. *Let S be a set of random keys from $\mathcal{K}$ and $V_{S,e} \subseteq S$ the subset of valid keys for a given ciphertext e i.e., the keys in S such that $D(k, e) \neq \perp$.*

We say (E, D) satisfies robustness *if for all ciphertexts $e \leftarrow \mathcal{A}(1^\kappa, S)$ adversarially generated by a PPT $\mathcal{A}$, $|V_{S,e}| \leq 1$ except with negligible probability.*

The definition says that it should be hard for an adversary to generate a ciphertext which can be decrypted to more than one valid ciphertext using any polynomial number of randomly generated keys (even for adversaries who see those keys before generating the ciphertext).

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 a very simple scheme which satisfies Definition 1 and 2: let $\mathcal{M} = \{0,1\}^\ell$ and $\mathcal{K} = \mathcal{C} = \{0,1\}^{\ell+\kappa}$. The encryption algorithm $E(k, m)$ parses k as (α, β) and $e = (m \oplus \alpha, \beta)$. The decryption algorithm $D(k, e)$ parses $k = (\alpha, \beta)$ and $e = (e_1, e_2)$ and outputs $\perp$ if $e_2 \neq \beta$ or outputs $m = e_1 \oplus \alpha$ otherwise. It can be shown that:

Lemma 3. *The scheme (E, D) defined above satisfies Definition 1 and 2.*

Proof. We show that the scheme satisfies Definition 1 and 2 in a strong, information theoretic sense. For Definition 1: $\mathcal{S}_1$ outputs a random $e \leftarrow \{0,1\}^{\ell+\kappa}$; $\mathcal{S}_2(e, M)$ parses $e = (e_1, e_2)$ and outputs $k = (e_1 \oplus M, e_2)$. The simulated distribution is trivially identical to the real one. For Definition 2: given any ciphertext $e = (e_1, e_2)$, $D((\alpha, \beta), e) \neq \perp$ implies that $\beta = e_2$. Thus even an unbounded adversary can break robustness of the scheme only if there are two keys $k_i, k_j \in S$ such that $\beta_i = \beta_j$ which only happens with probability negligible in κ.

Non-Malleability in Practice. When instantiating our protocol we must replace the random oracle with a hash function. To approximate the model, one can "localize" the random oracle by prepending the parties *id*'s and the session *id* to the hash function. We argue here that our choice of using the transcript of the protocol (S, R^i) as salt for the hash function helps in making sure that the oracle is *local* to the protocol, and helps against malleability attacks in cases where the parties' and session *id*'s are unavailable. Consider the following man-in-the middle attack, where an adversary $\mathcal{A}$ plays two copies of the $\binom{n}{1}$-OT, one as 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 the $\{e_i\}_{i\in[n]}$ from $\mathcal{S}$ and sets $e'_i = e_{(i-1 \bmod n)}$ to $\mathcal{R}$. It is easy to see that if the same hash function is used to instantiate the random oracle in the two protocols (and if $c \neq 0$), then the honest receiver outputs $z = M_{c+1}$, which is clearly a breach of security (i.e., this attack could not be run if the protocols are replaced with OT functionalities).

The previous attack can be seen as a malleability attack on the choice bit. An adversary can also try a malleability attack on the sender messages by forwarding $(S', R') = (S, R)$ but then manipulating the e_i's into ciphertexts e'_i which decrypt to related messages. In the $\binom{2}{1}$-OT, these attacks can be mitigated by using

authenticated encryption for (E, D) (which also satisfies *robustness* as in Definition 2). Now an adversary who changes both ciphertext is equivalent to an ideal adversary using input $(\perp, \perp)$, while an adversary who only changes one ciphertext, say e_c, is equivalent to an adversary which uses input bit $1 - c$ on the left and inputs $(m_{1-c}, \perp)$ on the right. This attack could also be run in an idealized world where parties have access to an OT functionality. Unfortunately for $\binom{n}{1}$-OT (with $n > 2$) this is not the case, since the protocol allows to "copy" any subset of messages, which would not be possible in an idealized setting.

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[4]). 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 $(X : Y : Z : T)$ 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 $r \in \mathbb{Z}_p$ it is efficient to compute $\mathcal{E}_X(rB)$, and given r and $\mathcal{E}_X(P)$ it is efficient to compute $\mathcal{E}_X(rP)$.

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

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

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

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 $32 + 32m$ 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 $\mathcal{E}_0(c^i(S + t) + x^i B) = \mathcal{E}_0(c^i t + R^i)$; 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 $\mathcal{E}_X(8x^i B)$ 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 $\mathcal{E}^{(1)}(R^i) = \mathcal{E}_0(8R^i + t)$; 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 $x^i S, yR^i$, or $yR^i - jT$ for $j \in [n]$. This is implemented by hashing $\mathcal{E}_1(S) \parallel \mathcal{E}_2(R^i) \parallel \mathcal{E}_2(P)$ 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., $n = 2$).

4 Field Arithmetic

This section describes our implementation strategy for arithmetic operations in $\mathbb{F}_{2^{255}-19}$, which serve as low-level building blocks for operations on the curve. 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 serial 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,

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

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

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.

which is called *vectorization*, makes our vectorized implementation achieve much higher throughtput 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

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.

the following 3 carry chains:

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

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.

Field inversion is implemented as a fix sequence of field squarings and multiplications.

Vectorization. We decompose field operations 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].

5 Implementation Results

This section compares the speed of our implementation of $\binom{2}{1}$-OT (i.e., $n = 2$) with other similar implementations. We stress that our software is a constant-time one: timing attacks are avoided using the same high-level strategy as [BDL+11].

To show that our speeds for curve operations are competitive, we modify the software to support the function of Diffie-Hellman key exchange and compare the results with existing Curve25519 implementations (our implementation performs scalar multiplications on the twisted Edwards curve, so it is not the same as Curve25519). 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. Note that all experiments are performed with Turbo Boost disabled.

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 Curve25519 implementation is the 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 Curve25519 implementation is from [BDL+11]. Note that since our software runs 4 scalar multiplications

		h6sandy	h9ivy
[MF15]	Average cycles to compute a public key	61828	57612
[BDL+11]	Average cycles to compute a shared secret	194036	182708
this work	Average cycles to generate a public key	61458	60853
	Average cycles to compute a shared secret	182169	180343

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

	m	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.

in parallel, the numbers in the table are the time for generating 4 public keys or 4 shared secrets divided by 4. In other words, our implementation is optimized for *througput* instead of *latency*.

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 $m \in \{4, 8, \ldots, 1024\}$.

Memory consumption. Our code for public-key generation uses a 284-KB table. For shared-secret computation the table size is 12 KB. For OTs, $\mathcal{S}$ uses a 12-KB table, while $\mathcal{R}$ is *allowed* to use a table of size up to 1344 KB which depends on the parameters given. The current code provides 4 copies of the precomputed points, one for each of the 4 scalar multiplcations, so it is possible to reduce the table sizes by a factor of 4 by broadcasting the precomputed points. Another reason that we have large tables is because of the representation for field elements: each limbs takes 8 bytes, so each field element already takes $12 \cdot 8 = 96$ bytes. The window sizes we use are the same as [BDL+11]. See [BDL+11] for issues related to table sizes.

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; the anonymous LATINCRYPT reviewer and in particular Gregory Neven.

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.

BCP04. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. New security results on encrypted key exchange. In *Public Key Cryptography - PKC 2004, 7th International Workshop on Theory and Practice in Public Key Cryptography, Singapore, March 1-4, 2004*, pages 145–158, 2004.

BDD+17. Paulo S. L. M. Barreto, Bernardo David, Rafael Dowsley, Kirill Morozov, and Anderson C. A. Nascimento. A framework for efficient adaptively secure composable oblivious transfer in the rom. Cryptology ePrint Archive, Report 2017/993, 2017. `https://eprint.iacr.org/2017/993`, Version 21 December 2017.

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`.

BLN+15. Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. `http://eprint.iacr.org/`.

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.

BPRS17. Megha Byali, Arpita Patra, Divya Ravi, and Pratik Sarkar. Fast and universally-composable oblivious transfer and commitment scheme with adaptive security. Cryptology ePrint Archive, Report 2017/1165, 2017. `https://eprint.iacr.org/2017/1165`, Version 21 Mar 2018.

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.

CJS14. Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 597–608, 2014.

CO15. Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, pages 40–58, 2015.

DKLs18. Jack Doerner, Yashvanth Kondi, Eysa Lee, and shelat abhi. Secure two-party threshold ecdsa from ecdsa assumptions. IEEE Security and Privacy Symposium, Cryptology ePrint Archive, Report 2018/499, 2018. `https://eprint.iacr.org/2018/499`, Version 23 May 2018.

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`.

GIR17. Ziya Alper Genç, Vincenzo Iovino, and Alfredo Rial. "the simplest protocol for oblivious transfer" revisited. Cryptology ePrint Archive, Report 2017/370, 2017. `https://eprint.iacr.org/2017/370`, Version 24 May 2017.

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

HL17. Eduard Hauck and Julian Loss. Efficient and universally composable protocols for oblivious transfer from the cdh assumption. Cryptology ePrint Archive, Report 2017/1011, 2017. `https://eprint.iacr.org/2017/1011`, Version 24 October 2017.

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.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. CRYPTO, 2015.

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

LM18. Baiyu Li and Daniele Micciancio. Equational security proofs of oblivious transfer protocols. In *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, pages 527–553, 2018.

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.