A New Approach to Practical Active-Secure Two-Party Computation[⋆]

Jesper Buus Nielsen[1,⋆⋆,⋆⋆⋆], Peter Sebastian Nordholt[1,⋆⋆], Claudio Orlandi[2,†], Sai Sheshank Burra[3]

[1] Aarhus University
[2] Bar-Ilan University
[3] Indian Institute of Technology Guwahati

Abstract. We propose a new approach to practical two-party computation secure against an active adversary. All prior practical protocols were based on Yao's garbled circuits. We use an OT-based approach and get efficiency via OT extension in the random oracle model. To get a practical protocol we introduce a number of novel techniques for relating the outputs and inputs of OTs in a larger construction.

We also report on an implementation of this approach, that shows that our protocol is more efficient than any previous one: For big enough circuits, we can evaluate more than 20000 Boolean gates per second. As an example, evaluating one oblivious AES encryption (~ 34000 gates) takes 64 seconds, but when repeating the task 27 times it only takes less than 3 seconds per instance.

1 Introduction

Secure two-party computation (2PC), introduced by Yao [32], allows two parties to jointly compute any function of their inputs in such a way that 1) the output of the computation is correct and 2) the inputs are kept private. Yao's protocol is secure only if the participants are *semi-honest* (they follow the protocol but try to learn more than they should by looking at their transcript of the protocol). A more realistic security definition considers *malicious adversaries*, that can arbitrarily deviate from the protocol.

A large number of approaches to 2PC have been proposed, falling into three main types, those based on Yao's garbled circuit techniques, those based on some form of homomorphic encryption and those based on oblivious transfer. Recently a number of efforts to implement 2PC in practice have been reported on; In sharp contrast to the theory, almost all of these are based on Yao's garbled circuit technique. A main advantage of Yao's garbled circuits is that it is primarily based on symmetric primitives: It uses one OT per input bit, but then uses only a few calls to, e.g., a hash function

⋆ A full version of this article can be found at `http://eprint.iacr.org/2011/091`.

⋆⋆ Partially supported by the Danish National Research Foundation and the National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Theory of Interactive Computation.

⋆⋆⋆ Partially supported by a Sapere Aude grant from the Danish Council for Independent Research.

† Supported by the European Research Council as part of the ERC project LAST.

per gate in the circuit to be evaluated. The other approaches are heavy on public-key primitives which are typically orders of magnitude slower than symmetric primitives.

However, in 2003 Ishai *et al.* introduced the idea of extending OTs *efficiently* [18]—their protocol allows to turn κ seed OTs based on public-key crypto into any polynomial $\ell = \mathrm{poly}(\kappa)$ number of OTs using only $O(\ell)$ invocations of a cryptographic hash function. For big enough ℓ the cost of the κ seed OTs is amortized away and OT extension essentially turns OT into a symmetric primitive in terms of its computational complexity. Since the basic approach of basing 2PC on OT in [14] is efficient in terms of consumption of OTs and communication, this gives the hope that OT-based 2PC too could be practical. This paper reports on the first implementation made to investigate the practicality of OT-based 2PC.

Our starting point is the efficient passive-secure OT extension protocol of [18] and passive-secure 2PC of [14]. In order to get active security and preserve the high practical efficiency of these protocols we chose to develop substantially different techniques, differentiating from other works that were only interested in *asymptotic* efficiency [15,29,20]. We report a number of contributions to the theory and practice of 2PC:

1. We introduce a new technical idea to the area of extending OTs efficiently, which allows to dramatically improve the practical efficiency of active-secure OT extension. Our protocol has the same asymptotic complexity as the previously best protocol in [15], but it is only a small factor slower than the passive-secure protocol in [18].
2. We give the first implementation of the idea of extending OTs efficiently with active security. The protocol generates $500{,}000$ OTs per second, showing that implementations needing a large number of actively secure OTs can be practical.
3. We introduce new technical ideas which allow to relate the outputs and inputs of OTs in a larger construction, via the use of information theoretic tags. This can be seen as a new flavor of committed OT that only requires symmetric cryptography. In combination with our first contribution, our protocol shows how to efficiently extend committed OT. Our protocols assume the existence of OT and are secure in the random oracle model.
4. We give the first implementation of practical 2PC not based on Yao's garbled circuit technique. Introducing a new practical technique is a significant contribution to the field in itself. In addition, our protocol shows favorable timings compared to the Yao-based implementations.

1.1 Comparison with Related Work

The question on the *asymptotic* computational overhead of cryptography was (essentially) settled in [19]. On the other hand, there is a growing interest in understanding the *practical* overhead of secure computation, and several works have perfected and implemented protocols based on Yao's garbled circuits [28,3,26,23,30,31,16,27,25,1,17], protocols based on homomorphic encryption [21,10,22,4] and protocols based on OT [20,24,6].

		Security	Model	Rounds	Time
(a)	DK [9] (3 parties)	Passive	SM	$O(d)$	1.5s
(b)	DK [9] (4 parties)	Active	SM	$O(d)$	4.5s
(c)	sS [1]	Active	SM	$O(1)$	192s
(d)	HEKM [17]	Passive	ROM	$O(1)$	0.2s
(e)	IPS-LOP [20,24]	Active	SM	$O(d)$	79s
(f)	This (single)	Active	ROM	$O(d)$	64s
(g)	This (27, amortized)	Active	ROM	$O(d)$	2.5s

Table 1. Brief comparison with other implementations.

A brief comparison of the time needed for oblivious AES evaluation for the best known implementations are shown in Table 1.[4] The protocols in rows (a-b) are for 3 and 4 parties respectively, and are secure against at most one corrupted party. One of the goals of the work in row (c) is how to efficiently support different outputs for different parties: in our OT based protocol this feature comes for free. The time in row (e) is an estimate made by [24] on the running time of their optimized version of the OT-based protocol in [20]. The column *Round* indicates the round complexity of the protocols, d being the depth of the circuit while the column *Model* indicates whether the protocol was proven secure in the standard model (SM) or the random oracle model (ROM).

The significance of this work is shown in row (g). The reason for the dramatic drop between row (f) and (g) is that in (f), when we only encrypt one block, our implementation preprocesses for many more gates than is needed, for ease of implementation. In (g) we encrypt 27 blocks, which is the minimum value which eats to up all the preprocessed values. We consider these results positive: our implementation is as fast or faster than any other 2PC protocol, even when encrypting only one block. And more importantly, when running at full capacity, the price to pay for active security is about a factor 10 against the passive-secure protocol in (d). We stress that this is only a limited comparison, as the different experiments were run on different hardware and network setups: when several options were available, we selected the best time reported by the other implementations. See Sect. 6 for more timings and details of our implementation.

1.2 Overview of Our Approach

We start from a classic textbook protocol for 2PC [13, Sec. 7.3]. In this protocol, Alice holds secret shares x_A, y_A and Bob holds secret shares x_B, y_B of some bits x, y s.t. $x_A \oplus x_B = x$ and $y_A \oplus y_B = y$. Alice and Bob want to compute secret shares of $z = g(x, y)$ where g is some Boolean gate, for instance the AND gate: Alice and Bob need to compute a random sharing z_A, z_B of $z = xy = x_A y_A \oplus x_A y_B \oplus x_B y_A \oplus x_B y_B$. The parties can compute the AND of their local shares ($x_A y_A$ and $x_B y_B$), while they can use oblivious transfer (OT) to compute the cross products ($x_A y_B$ and $x_B y_A$). Now the parties can iterate for the next layer of the circuit, up to the end where they will reconstruct the output values by revealing their shares.

[4] Oblivious AES has become one of the most common circuits to use for benchmarking generic MPC protocols, due to its reasonable size (about 30000 gates) and its relevance as a building block for constructing specific purpose protocols, like private set intersection [11].

This protocol is secure against a semi-honest adversary: assuming the OT protocol to be secure, Alice and Bob learn nothing about the intermediate values of the computation. It is easy to see that if a large circuit is evaluated, then the protocol is not secure against a malicious adversary: any of the two parties could replace values on any of the internal wires, leading to a possibly incorrect output and/or leakage of information.

To cope with this, we put MACs on all bits. The starting point of our protocol is *oblivious authentication* of bits. One party, the *key holder*, holds a uniformly random *global key* $\Delta \in \{0,1\}^\kappa$. The other party, the *MAC holder*, holds some secret bits (x, y, say). For each such bit the key holder holds a corresponding uniformly random *local key* ($K_x, K_y \in \{0,1\}^\kappa$) and the MAC holder holds the corresponding *MAC* ($M_x = K_x \oplus x\Delta$, $M_y = K_y \oplus y\Delta$). The key holder does not know the bits and the MAC holder does not know the keys. Note that $M_x \oplus M_y = (K_x \oplus K_y) \oplus (x \oplus y)\Delta$. So, the MAC holder can locally compute a MAC on $x \oplus y$ under the key $K_x \oplus K_y$ which is non-interactively computable by the key holder. This homomorphic property comes from fixing Δ and we exploit it throughout our constructions. From a bottom-up look, our protocol is constructed as follows (see Fig. 1 for the main structure):

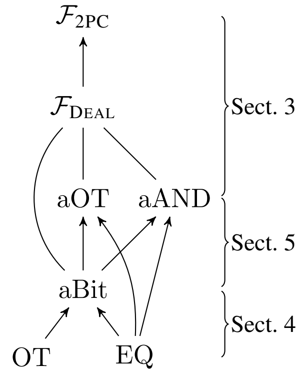

Fig. 1. Paper outline. This order of presentation is chosen to allow the best progression in introduction of our new techniques.

Bit Authentication: We first implement oblivious authentication of bits (aBit). As detailed in Sect. 4, to construct authenticated bits we start by extending a few (say $\kappa = 640$) seed $\binom{2}{1}$-OTs into many (say $\ell = 2^{20}$) OTs, using OT extension. Then, if A wants to get a bit x authenticated, she can input it as the choice bit in an OT, while B can input $(K_x, K_x \oplus \Delta)$, playing the sender in the OT. Now A receives $M_x = K_x \oplus x\Delta$. It should, of course, be ensured that even a corrupted B uses the same value Δ in all OTs. I.e., it should hold for all produced OTs that the XORs of the offered message pairs are constant—this constant value is then taken to be Δ. It turns out, however, that when using the highly efficient *passive-secure* OT extender in [18] and starting from seed OTs where the XORs of message pairs are constant, one also produces OTs where the XORs of message pairs are constant, and we note that for this use the protocol in [18] happens to be *active-secure*! Using cut-and-choose we ensure that most of the XORs of message pairs offered in the seed OTs are constant, and with a new and inexpensive trick we offer privacy and correctness even if few of these XORs have different values. This cut-and-choose technique uses one call to a box EQ for checking equality.

Authenticated local AND: From aBits we then construct *authenticated local AND*s (aAND), where the MAC holder locally holds random authenticated bits a, b, c with $c = ab$. To create authenticated local ANDs, we let one party compute $c = ab$ for random a and b and get authentications on a, b, c (when creating aANDs, we assume the aBits are already available). The challenge is to ensure that $c = ab$. We construct an efficient proof for this fact, again using the box EQ once. This proof might, however, leak the bit a with small but noticeable probability. We correct this using a combiner.

Authenticated OT: From aBits we also construct *authenticated OT*s (aOT), which are normal $\binom{2}{1}$-OTs of bits, but where all input bits and output bits are obliviously authenticated. This is done by letting the two parties generate aBits representing the sender messages x_0, x_1 and the receiver choice bit c. To produce the receiver's output, first a random aBit is sampled. Then this bit is "corrected" in order to be consistent with the run of an OT protocol with input messages x_0, x_1 and choice bit c. This correction might, however, leak the bit c with small but noticeable probability. We correct this using an OT combiner. One call to the box EQ is used.

2PC: Given two aANDs and two aOTs one can evaluate in a very efficient way any Boolean gate: only 4 bits per gate are communicated, as the MACs can be checked in an amortized manner.

That efficient 2PC is possible given enough aBits, aANDs and aOTs is no surprise. In some sense, it is the standard way to base passive-secure 2PC on passive-secure OT enhanced with a particular flavor of committed OT (as in [8,12]). What is new is that we managed to find a particular committed OT-like primitive which allows both a very efficient generation and a very efficient use: while previous results based on committed OT require hundreds of *exponentiations* per gate, our cost per gate is in the order of hundreds of *hash functions*. To the best of our knowledge, we present the first practical approach to extending a few seed OTs into a large number of committed OT-like primitives. Of more specific technical contributions, the main is that we manage to do all the proofs efficiently, thanks also to the preprocessing nature of our protocol: Creating aBits, we get active security paying only a constant overhead over the passive-secure protocol in [18]. In the generation of aANDs and aOTs, we replace cut-and-choose with efficient, slightly leaky proofs and then use a combiner to get rid of the leakage: When we preprocess for ℓ gates and combine B leaky objects to get each potentially unleaky object, the probability of leaking is $(2\ell)^{-B} = 2^{-\log_2(\ell)(B-1)}$. As an example, if we preprocess for 2^{20} gates with an overhead of $B = 6$, then we get leakage probability 2^{-100}.

As a corollary to being able to generate any $\ell = \mathrm{poly}(\kappa)$ active-secure aBits from $O(\kappa)$ seed OTs and $O(\ell)$ calls to a hash-function, we get that we can generate any $\ell = \mathrm{poly}(\kappa)$ active-secure $\binom{2}{1}$-OTs of κ-bit strings from $O(\kappa)$ seed OTs and $O(\ell)$ calls to a hash-function, matching the asymptotic complexity of [15] while dramatically reducing their hidden constants.

2 Preliminaries and Notation

We use κ (and sometimes ψ) to denote the security parameter. We require that a poly-time adversary break the protocol with probability at most $\mathrm{poly}(\kappa)2^{-\kappa}$. For a bit-string $S \in \{0,1\}^*$ we define $0S \overset{\text{def}}{=} 0^{|S|}$ and $1S \overset{\text{def}}{=} S$. For a finite set S we use $s \in_{\text{R}} S$ to denote that s is chosen uniformly at random in S. For a finite distribution D we use $x \leftarrow D$ to denote that x is sampled according to D.

The UC Framework. We prove our results static, active-secure in the UC framework [5], and we assume the reader to be familiar with it. We will idiosyncratically use the word *box* instead of the usual term *ideal functionality*. To simplify the statements of our results we use the following terminology:

Definition 1. *We say that a box* A *is* reducible *to a box* B *if there exist an actively secure implementation* π *of* A *which uses only one call to* B. *We say that* A *is* locally *reducible to* B *if the parties of* π *do not communicate (except through the one call to* B*). We say that* A *is* linear *reducible to* B *if the computing time of all parties of* π *is linear in their inputs and outputs. We use* equivalent *to denote reducibility in both directions.*

It is easy to see that if A is (linear, locally) reducible to B and B is (linear, locally) reducible to C, then A is (linear, locally) reducible to C.

Hash Functions. We use a hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$, which we model as a random oracle (RO). We sometimes use H to mask a message, as in $H(x) \oplus M$. If $|M| \neq \kappa$, this denotes $\mathrm{prg}(H(x)) \oplus M$, where prg is a pseudo-random generator $\mathrm{prg} : \{0,1\}^\kappa \to \{0,1\}^{|M|}$. We also use a collision-resistant hash function $G : \{0,1\}^{2\kappa} \to \{0,1\}^\kappa$.

As other 2PC protocols whose focus is efficiency [23,17], we are content with a proof in the random oracle model. What is the exact assumption on the hash function that we need for our protocol to be secure, as well as whether this can be implemented under standard cryptographic assumption is an interesting theoretical question, see [2,7].

Oblivious Transfer. We use a box $\mathrm{OT}(\tau, \ell)$ which can be used to perform τ $\binom{2}{1}$-oblivious transfers of strings of bit-length ℓ. In each of the τ OTs the sender S has two inputs $x_0, x_1 \in \{0,1\}^\ell$, called the *messages*, and the receiver R has an input $c \in \{0,1\}$, called the *choice bit*. The output to R is $x_c = c(x_0 \oplus x_1) \oplus x_0$. No party learns any other information.

Equality Check. We use a box $\mathrm{EQ}(\ell)$ which allows two parties to check that two strings of length ℓ are equal. If they are different the box leaks both strings to the adversary, which makes secure implementation easier. We define and use this box to simplify the exposition of our protocol. In practice we implement the box using the commit-and-open approach. Hashing a string (together with some randomness) is a secure implementation of commitment in the random oracle model. See the full version for more details.

Leakage Functions. We use a concept of a *leakage function on τ bits*, which is a class $\mathcal{L}$ of distributions, where each $L \in \mathcal{L}$ is a distribution on $(S, c) \in 2^{\{1,\ldots,\tau\}} \times \{0,1\}$, where, as we will see later, $c = 0$ is interpreted as a failure to create leakage and $c = 1$ is interpreted as leakage of the bits indexed by $i \in S$. We say that L is κ-secure if the expected value of $\tau - c|S|$ is at least κ and we says that $\mathcal{L}$ is κ-secure if all $L \in \mathcal{L}$ are κ-secure. See the full version for a more detailed definition.

3 The Two-Party Computation Protocol

We want to implement the box $\mathcal{F}_{\text{2PC}}$ for Boolean two-party secure computation as described in Fig. 4. We will implement this box in the $\mathcal{F}_{\text{DEAL}}$-hybrid model of Fig. 5. This box provides the parties with aBits, aANDs and aOTs, and models the preprocessing phase of our protocol. In Fig. 3 we introduce notation for working with authenticated bits. The protocol implementing $\mathcal{F}_{\text{2PC}}$ in the dealer model is described in Fig. 6. The dealer offers random authenticated bits (to A or B), random authenticated local AND triples and random authenticated OTs. Those are all the ingredients that we need to build the 2PC protocol. Note that the dealer offers randomized versions of all commands: this is not a problem as the "standard" version of the commands (the one where the parties can specify their input bits instead of getting them at random from the box) are linearly reducible to the randomized version, as can be easily deduced from the protocol description. The following result is proven in the full version.

$$\begin{tikzcd}
 & \mathcal{F}_{\text{2PC}} & \\
 & \mathcal{F}_{\text{DEAL}} \arrow[u] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\mathrm{aBit} & \mathrm{aOT} & \mathrm{aAND}
\end{tikzcd}$$

Fig. 2. Sect. 3 outline.

Theorem 1. *The protocol in Fig. 6 securely implements the box $\mathcal{F}_{\text{2PC}}$ in the $\mathcal{F}_{\text{DEAL}}$-hybrid model with security parameter κ.*

Why the global key queries? The $\mathcal{F}_{\text{DEAL}}$ box (Fig. 5) allows the adversary to guess the value of the global key, and it informs it if its guess is correct. This is needed for technical reasons: When $\mathcal{F}_{\text{DEAL}}$ is proven UC secure, the environment has access to either $\mathcal{F}_{\text{DEAL}}$ or the protocol implementing $\mathcal{F}_{\text{DEAL}}$. In both cases the environment learns the global keys Δ_A and Δ_B. In particular, the environment learns Δ_A even if B is honest. This requires us to prove the sub-protocol for $\mathcal{F}_{\text{DEAL}}$ secure to an adversary knowing Δ_A even if B is honest: to be be able to do this, the simulator needs to recognize Δ_A if it sees it—hence the global key queries. Note, however, that in the context where we use $\mathcal{F}_{\text{DEAL}}$ (Fig. 6), the environment does *not* learn the global key Δ_A when B is honest: A corrupted A only sees MACs on one bit using the same local key, so all MACs are uniformly random in the view of a corrupted A, and B never makes the local keys public.

Amortized MAC checks. In the protocol of Fig. 6, there is no need to send MACs and check them every time we do a "reveal". In fact, it is straightforward to verify that before an **Output** command is executed, the protocol is perfectly secure even if the MACs are not checked. Notice then that a key holder checks a MAC M_x on a bit x by computing $M'_x = K_x \oplus x\Delta$ and comparing M'_x to the M_x which was sent along with x. These equality checks can be deferred and amortized. Initially the MAC holder, e.g. A, sets $N = 0^\kappa$ and the key holder, e.g. B, sets $N' = 0^\kappa$. As long as no **Output** command is executed, when A reveals x she updates $N \leftarrow G(N, H(M_x))$ for the MAC M_x she should have sent along with x, and B updates $N' \leftarrow G(N', H(M'_x))$; Here G is a collision resistant hash function. Before executing an **Output**, A sends N to B who aborts if $N \neq N'$. Security of this check is easily proved in the random oracle model. The optimization brings the communication complexity of the protocol down

Global Key: We call $\Delta_A, \Delta_B \in \{0,1\}^\kappa$ the two *global keys*, held by B and A respectively.

Authenticated Bit: We write $[x]_\mathsf{A}$ to represent an *authenticated secret bit* held by A. Here B knows a key $K_x \in \{0,1\}^\kappa$ and A knows a bit x and a MAC $M_x = K_x \oplus x\Delta_A \in \{0,1\}^\kappa$. Let $[x]_\mathsf{A} \stackrel{\text{def}}{=} (x, M_x, K_x)$.[a]

If $[x]_\mathsf{A} = (x, M_x, K_x)$ and $[y]_\mathsf{A} = (y, M_y, K_y)$ we write $[z]_\mathsf{A} = [x]_\mathsf{A} \oplus [y]_\mathsf{A}$ to indicate $[z]_\mathsf{A} = (z, M_z, K_z) \stackrel{\text{def}}{=} (x \oplus y, M_x \oplus M_y, K_x \oplus K_y)$. Note that no communication is required to compute $[z]_\mathsf{A}$ from $[x]_\mathsf{A}$ and $[y]_\mathsf{A}$.

It is possible to authenticate a constant bit (a value known both to A and B) $b \in \{0,1\}$ as follows: A sets $M_b = 0^\kappa$, B sets $K_b = b\Delta_A$, now $[b]_\mathsf{A} \stackrel{\text{def}}{=} (b, M_b, K_b)$. For a constant b we let $[x]_\mathsf{A} \oplus b \stackrel{\text{def}}{=} [x]_\mathsf{A} \oplus [b]_\mathsf{A}$, and we let $b[x]_\mathsf{A}$ be equal to $[0]_\mathsf{A}$ if $b = 0$ and $[x]_\mathsf{A}$ if $b = 1$.

We say that A *reveals* $[x]_\mathsf{A}$ by sending (x, M_x) to B who aborts if $M_x \neq K_x \oplus x\Delta_A$. Alternatively we say that A *announces* x by sending x to B without a MAC.

Authenticated bits belonging to B are written as $[y]_\mathsf{B}$ and are defined symmetrically, changing side of all the values and using the global value Δ_B instead of Δ_A.

Authenticated Share: We write $[x]$ to represent the situation where A and B hold $[x_A]_\mathsf{A}, [x_B]_\mathsf{B}$ and $x = x_A \oplus x_B$, and we write $[x] = ([x_A]_\mathsf{A}, [x_B]_\mathsf{B})$ or $[x] = [x_A|x_B]$.

If $[x] = [x_A|x_B]$ and $[y] = [y_A|y_B]$ we write $[z] = [x] \oplus [y]$ to indicate $[z] = ([z_A]_\mathsf{A}, [z_B]_\mathsf{B}) = ([x_A]_\mathsf{A} \oplus [y_A]_\mathsf{A}, [x_B]_\mathsf{B} \oplus [y_B]_\mathsf{B})$. Note that no communication is required to compute $[z]$ from $[x]$ and $[y]$.

It is possible to create an authenticated share of a constant $b \in \{0,1\}$ as follows: A and B create $[b] = [b|0]$. For a constant value $b \in \{0,1\}$, we define $b[x]$ to be equal to $[0]$ if $b = 0$ and $[x]$ if $b = 1$.

When an authenticated share is *revealed*, the parties reveal to each other their authenticated bits and abort if the MACs are not correct.

[a] Since Δ_A is a global value we will not always write it explicitly. Note that in $x\Delta_A$, x represents a *value*, 0 or 1, and that in $[x]_\mathsf{A}$, K_x and M_x it represents a *variable name*. I.e., there is only one key (MAC) per authenticated bit, and for the bit named x, the key (MAC) is named K_x (M_x). If $x = 0$, then $M_x = K_x$. If $x = 1$, then $M_x = K_x \oplus \Delta_A$.

Fig. 3. Notation for authenticated and shared bits.

Rand: On input (rand, vid) from A and B, with vid a fresh identifier, the box picks $r \in_\mathsf{R} \{0,1\}$ and stores (vid, r).

Input: On input $(\text{input}, \mathsf{P}, vid, x)$ from $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ and $(\text{input}, \mathsf{P}, vid, ?)$ from the other party, with vid a fresh identifier, the box stores (vid, x).

XOR: On command $(\text{xor}, vid_1, vid_2, vid_3)$ from both parties (if vid_1, vid_2 are defined and vid_3 is fresh), the box retrieves (vid_1, x), (vid_2, y) and stores $(vid_3, x \oplus y)$.

AND: As **XOR**, but store $(vid_3, x \cdot y)$.

Output: On input $(\text{output}, \mathsf{P}, vid)$ from both parties, with $\mathsf{P} \in \{\mathsf{A}, \mathsf{B}\}$ (and vid defined), the box retrieves (vid, x) and outputs it to P.

At each command the box leaks to the environment which command is being executed (keeping the value x in **Input** secret), and delivers messages only when the environment says so.

Fig. 4. The box $\mathcal{F}_{2\text{PC}}$ for Boolean Two-party Computation.

from $O(\kappa|C|)$ to $O(|C| + o\kappa)$, where o is the number of rounds in which outputs are opened. For a circuit of depth $O(|C|/\kappa)$, the communication is $O(|C|)$.

Initialize: On input (init) from A and (init) from B, the box samples $\Delta_A, \Delta_B \in \{0,1\}^\kappa$, stores them and outputs Δ_B to A and Δ_A to B. If A (resp. B) is corrupted, she gets to choose Δ_B (resp. Δ_A).

Authenticated Bit (A): On input $(\mathrm{aBIT}, \mathsf{A})$ from A and B, the box samples a random $[x]_\mathsf{A} = (x, M_x, K_x)$ with $M_x = K_x \oplus x\Delta_A$ and outputs it (x, M_x to A and K_x to B). If B is corrupted he gets to choose K_x. If A is corrupted she gets to choose (x, M_x), and the box sets $K_x = M_x \oplus x\Delta_A$.

Authenticated Bit (B): On input $(\mathrm{aBIT}, \mathsf{B})$ from A and B, the box samples a random $[x]_\mathsf{B} = (x, M_x, K_x)$ with $M_x = K_x \oplus x\Delta_B$ and outputs it (x, M_x to B and K_x to A). As in **Authenticated Bit (A)**, corrupted parties can choose their own randomness.

Authenticated local AND (A): On input $(\mathrm{aAND}, \mathsf{A})$ from A and B, the box samples random $[x]_\mathsf{A}$,$[y]_\mathsf{A}$ and $[z]_\mathsf{A}$ with $z = xy$ and outputs them. As in **Authenticated Bit (A)**, corrupted parties can choose their own randomness.

Authenticated local AND (B) Defined symmetrically.

Authenticated OT (A-B): On input $(\mathrm{aOT}, \mathsf{A}, \mathsf{B})$ from A and B, the box samples random $[x_0]_\mathsf{A}$,$[x_1]_\mathsf{A}$,$[c]_\mathsf{B}$ and $[z]_\mathsf{B}$ with $z = x_c = c(x_0 \oplus x_1) \oplus x_0$ and outputs them. As in **Authenticated Bit**, corrupted parties can choose their own randomness.

Authenticated OT (B-A): Defined symmetrically.[a]

Global Key Queries: The adversary can at any point input (A, Δ) and be told whether $\Delta = \Delta_B$. And it can at any point input (B, Δ) and be told whether $\Delta = \Delta_A$.

[a] The dealer offers aOTs in both directions. Notice that the dealer could offer aOT only in one direction and the parties could then "turn" them: as regular OT, aOT is symmetric as well.

Fig. 5. The box $\mathcal{F}_{\mathrm{DEAL}}$ for dealing preprocessed values.

Implementing $\mathcal{F}_{\mathrm{DEAL}}$. In the following sections we show how to implement $\mathcal{F}_{\mathrm{DEAL}}$. In Sect. 4 we implement just the part with the commands **Authenticated Bits**. In Sect. 5 we show how to extend with the **Authenticated OT** command, by showing how to implement many aOTs from many aBits. In the full version we show how to further extend with the **Authenticated local AND** command. We describe the extensions separately, but since they both maintain the value of the global keys, they will produce aANDs and aOTs with the same keys as the aBits used, giving an implementation of $\mathcal{F}_{\mathrm{DEAL}}$.

4 Bit Authentication

In this section we show how to efficiently implement (oblivious) bit authentication, i.e., we want to be in a situation where A knows some bits $x_1, \ldots, x_\ell$ together with MACs $M_1, \ldots, M_\ell$, while B holds a global key Δ_A and local keys $K_1, \ldots, K_\ell$ s.t. $M_i = K_i \oplus x_i\Delta_A$, as described in $\mathcal{F}_{\mathrm{DEAL}}$ (Fig. 5). Given the complete symmetry of $\mathcal{F}_{\mathrm{DEAL}}$, we only describe the case where A is MAC holder.

If the parties were honest, we could do the following: A and B run an OT where B inputs the two messages

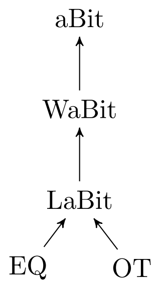
$$\begin{tikzcd}
 & \mathrm{aBit} & \\
 & \mathrm{WaBit} \arrow[u] & \\
 & \mathrm{LaBit} \arrow[u] & \\
\mathrm{EQ} \arrow[ur] & & \mathrm{OT} \arrow[ul]
\end{tikzcd}$$

Fig. 7. Sect. 4 outline.

Initialize: When activated the first time, A and B activate $\mathcal{F}_{\text{DEAL}}$ and receive Δ_B and Δ_A respectively.

Rand: A and B ask $\mathcal{F}_{\text{DEAL}}$ for random authenticated bits $[r_A]_{\mathsf{A}}, [r_B]_{\mathsf{B}}$ and stores $[r] = [r_A|r_B]$ under vid.

Input: If $\mathsf{P} = \mathsf{A}$, then A asks $\mathcal{F}_{\text{DEAL}}$ for an authenticated bit $[x_A]_{\mathsf{A}}$ and announces (i.e., no MAC is sent together with the bit) $x_B = x \oplus x_A$, and the parties build $[x_B]_{\mathsf{B}}$ and define $[x] = [x_A|x_B]$. The protocol is symmetric for B.

XOR: A and B retrieve $[x], [y]$ stored under vid_1, vid_2 and store $[z] = [x] \oplus [y]$ under vid_3. For brevity we drop explicit mentioning of variable identifiers below.

AND: A and B retrieve $[x], [y]$ and compute $[z] = [xy]$ as follows:

1. The parties ask $\mathcal{F}_{\text{DEAL}}$ for a random AND triplet $[u]_{\mathsf{A}}, [v]_{\mathsf{A}}, [w]_{\mathsf{A}}$ with $w = uv$.
   A reveals $[f]_{\mathsf{A}} = [u]_{\mathsf{A}} \oplus [x_A]_{\mathsf{A}}$ and $[g]_{\mathsf{A}} = [v]_{\mathsf{A}} \oplus [y_A]_{\mathsf{A}}$.
   The parties compute $[x_A y_A]_{\mathsf{A}} = f[y_A]_{\mathsf{A}} \oplus g[x_A]_{\mathsf{A}} \oplus [w]_{\mathsf{A}} \oplus fg$.
2. Symmetrically the parties compute $[x_B y_B]_{\mathsf{B}}$.
3. The parties ask $\mathcal{F}_{\text{DEAL}}$ for a random authenticated OT $[u_0]_{\mathsf{A}}, [u_1]_{\mathsf{A}}, [c]_{\mathsf{B}}, [w]_{\mathsf{B}}$ with $w = u_c$.
   They also ask for an authenticated bit $[r_A]_{\mathsf{A}}$.
   Now B reveals $[d]_{\mathsf{B}} = [c]_{\mathsf{B}} \oplus [y_B]_{\mathsf{B}}$.
   A reveals $[f]_{\mathsf{A}} = [u_0]_{\mathsf{A}} \oplus [u_1]_{\mathsf{A}} \oplus [x_A]_{\mathsf{A}}$ and $[g]_{\mathsf{A}} = [r_A]_{\mathsf{A}} \oplus [u_0]_{\mathsf{A}} \oplus d[x_A]_{\mathsf{A}}$.
   Compute $[s_B]_{\mathsf{B}} = [w]_{\mathsf{B}} \oplus f[c]_{\mathsf{B}} \oplus g$. Note that at this point $[s_B]_{\mathsf{B}} = [r_A \oplus x_A y_B]_{\mathsf{B}}$.
4. Symmetrically the parties compute $[s_A]_{\mathsf{A}} = [r_B \oplus x_B y_A]_{\mathsf{A}}$.

A and B compute $[z_A]_{\mathsf{A}} = [r_A]_{\mathsf{A}} \oplus [s_A]_{\mathsf{A}} \oplus [x_A y_A]_{\mathsf{A}}$ and $[z_B]_{\mathsf{B}} = [r_B]_{\mathsf{B}} \oplus [s_B]_{\mathsf{B}} \oplus [x_B y_B]_{\mathsf{B}}$ and let $[z] = [z_A|z_B]$.

Output: The parties retrieve $[x] = [x_A|x_B]$. If A is to learn x, B reveals x_B. If B is to learn x, A reveals x_A.

Fig. 6. Protocol for $\mathcal{F}_{\text{2PC}}$ in the $\mathcal{F}_{\text{DEAL}}$-hybrid model

$(K_i, K_i \oplus \Delta_A)$ and A inputs choice bit x_i, to receive $M_i = K_i \oplus x_i \Delta_A$. However, if B is dishonest he might not use the same Δ_A in all OTs. The main ideas that make the protocol secure against cheating parties are the following:

1. For reasons that will be apparent later, we will actually start in the opposite direction and let B receive some authenticated bits y_i using an OT, where A is supposed to always use the same global key Γ_B. Thus an honest A inputs $(L_i, L_i \oplus \Gamma_B)$ in the OTs and B receives $N_i = L_i \oplus y_i \Gamma_B$. To check that A is playing honest in most OTs, the authenticated bits are arranged into pairs and a check is performed, which restricts A to cheat in at most a few OTs.
2. We then notice that what A gains by using different Γ_B's in a few OTs is no more than learning a few of B's bits y_i. We call this a leaky aBit, or LaBit.
3. We show how to turn this situation into an equivalent one where A (not B) receives authenticated random bits x_i (none of which leaks to B) under a "slightly insecure" global key Γ_A. The insecurity comes from the fact that the leakage of the y_i's turns into the leakage of a few bits of the global key Γ_A towards A. We call this an aBit with weak global key, or WaBit.
4. Using privacy amplification, we amplify the previous setting to a new one where A receives authenticated bits under a (shorter) fully secure global key Δ_A, where no

bits of $\varDelta_A$ are known to A, finally implementing the aBit command of the dealer box.

We will proceed in reverse order and start with step 4 in the previous description: we will start with showing how we can turn authenticated bits under an "insecure" global key $\varGamma_A$ into authenticated bits under a "secure" (but shorter) global key $\varDelta_A$.

4.1 Bit Authentication with Weak Global Key (WaBit)

We will first define the box providing bit authentication, but where some of the bits of the global key might leak. We call this box WaBit (bit authentication with weak global key) and we formally describe it in Fig. 8. The box $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$ outputs ℓ bits with keys of length τ. The box is also parameterized by a class $\mathcal{L}$ of leakage functions on τ bits. The box $\mathrm{aBit}(\ell, \psi)$ is the box $\mathrm{WaBit}^{\mathcal{L}}(\ell, \psi)$ where $\mathcal{L}$ is the class of leakage functions that never leak.

Honest Parties:
1. The box samples $\varGamma_A \in_{\mathsf{R}} \{0,1\}^\tau$ and outputs it to B.
2. The box samples and outputs $[x_1]_\mathsf{A}, \dots, [x_\ell]_\mathsf{A}$. Each $[x_i]_\mathsf{A} = (x_i, M_i', K_i') \in \{0,1\}^{1+2\tau}$ s.t. $M_i' = K_i' \oplus x_i \varGamma_A$.

Corrupted Parties:
1. If A is corrupted, then A may choose a leakage function $L \in \mathcal{L}$. Then the box samples $(S, c) \leftarrow L$. If $c = 0$ the box outputs `fail` to B and terminates. If $c = 1$, the box outputs $\{(i, (\varGamma_A)_i)\}_{i \in S}$ to A.
2. If A is corrupted, then A chooses the x_i and the M_i' and then $K_i' = M_i' \oplus x_i \varGamma_A$.
3. If B is corrupted, then B chooses $\varGamma_A$ and the K_i'.

Global Key Queries: The adversary can input $\varGamma$ and will be told if $\varGamma = \varGamma_A$.

Fig. 8. The box $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$ for Bit Authentication with Weak Global Key

1. The parties invoke $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$ with $\tau = \frac{22}{3}\psi$. The output to A is $((M_1', x_1), \dots, (M_\ell', x_\ell))$. The output to B is $(\varGamma_A, K_1', \dots, K_\ell')$.
2. B samples $\mathbf{A} \in_{\mathsf{R}} \{0,1\}^{\psi \times \tau}$, a random binary matrix with ψ rows and τ columns, and sends $\mathbf{A}$ to A.
3. A computes $M_i = \mathbf{A} M_i' \in \{0,1\}^\psi$ and outputs $((M_1, x_1), \dots, (M_\ell, x_\ell))$.
4. B computes $\varDelta_A = \mathbf{A}\varGamma_A$ and $K_i = \mathbf{A} K_i'$ and outputs $(\varDelta_A, K_1, \dots, K_\ell)$.

Fig. 9. Sub-protocol for reducing $\mathrm{aBit}(\ell, \psi)$ to $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$.

In Fig. 9 we describe a protocol which takes a box WaBit, where one quarter of the bits of the global key might leak, and amplifies it to a box aBit where the global key is perfectly secret. The protocol is described for general $\mathcal{L}$ and it is parameterized by a desired security level ψ. The proof of the following theorem can be found in the full version.

Honest Parties:

1. The box samples $\Gamma_B \in_R \{0,1\}^\ell$ and outputs it to A.
2. The box samples and outputs $[y_1]_B, \ldots, [y_\tau]_B$. Each $[y_i]_B = (y_i, N_i, L_i) \in \{0,1\}^{1+2\ell}$ s.t. $N_i = L_i \oplus y_i \Gamma_B$.

Corrupted Parties:

1. If A is corrupted, then A may input a leakage function $L \in \mathcal{L}$. Then the box samples $(S, c) \leftarrow L$. If $c = 0$ the box outputs `fail` to B and terminates. If $c = 1$, the box outputs $\{(i, y_i)\}_{i \in S}$ to A.
2. Corrupted parties get to specify their outputs as in Fig. 8.

Choice Bit Queries: The adversary can input Δ and will be told if $\Delta = (y_1, \ldots, y_\tau)$.

Fig. 10. The box $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$ for Bit Authentication with Leaking Bits

1. A and B invoke $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$. B learns $((N_1, y_1), \ldots, (N_\tau, y_\tau))$ and A learns $(\Gamma_B, L_1, \ldots, L_\tau)$.
2. A lets x_j be the j-th bit of Γ_B and M_j the string consisting of the j-th bits from all the strings L_i, i.e. $M_j = L_{1,j}||L_{2,j}||\ldots||L_{\ell,j}$.
3. B lets Γ_A be the string consisting of all the bits y_i, i.e. $\Gamma_A = y_1||y_2||\ldots||y_\ell$, and lets K_j be the string consisting of the j-th bits from all the strings N_i, i.e. $K_j = N_{1,j}||N_{2,j}||\ldots||N_{\ell,j}$.
4. A and B now hold $[x_j]_A = (x_j, M_j, K_j)$ for $j = 1, \ldots, \ell$.

Fig. 11. Sub-protocol for reducing $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$ to $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$

Theorem 2. *Let $\tau = \frac{22}{3}\psi$ and $\mathcal{L}$ be a $\left(\frac{3}{4}\tau\right)$-secure leakage function on τ bits. The protocol in Fig. 9 securely implements $\mathrm{aBit}(\ell, \psi)$ in the $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$-hybrid model with security parameter ψ. The communication is $O(\psi^2)$ and the work is $O(\psi^2 \ell)$.*

4.2 Bit Authentication with Leaking Bits (LaBit)

We now show another insecure box for aBit. The new box is insecure in the sense that a few of the bits to be authenticated might leak to the other party. We call this box an aBit with leaking bits, or LaBit and formally describe it in Fig. 10. The box $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$ outputs τ authenticated bits with keys of length ℓ, and is parameterized by a class of leakage functions $\mathcal{L}$ on τ-bits. We show that $\mathrm{WaBit}^{\mathcal{L}}$ can be reduced to $\mathrm{LaBit}^{\mathcal{L}}$. In the reduction, a LaBit that outputs authenticated bits $[y_i]_B$ to B is turned into a WaBit that outputs authenticated bits $[x_j]_A$ to A, therefore we present the LaBit box that outputs bits to B. The reduction is strongly inspired by the OT extension techniques in [18].

Theorem 3. *For all ℓ, τ and $\mathcal{L}$ the boxes $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$ and $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$ are linear locally equivalent, i.e., can be implemented given the other in linear time without interaction.*

The proof can be found in the full version. Note that since we turn $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$ into $\mathrm{WaBit}^{\mathcal{L}}(\ell, \tau)$, if we choose $\ell = \mathrm{poly}(\psi)$ we can turn a relatively small number ($\tau = \frac{22}{3}\psi$) of authenticated bits towards one player into a very larger number (ℓ) of authenticated bits towards the other player.

4.3 A Protocol For Bit Authentication With Leaking Bits

In this section we show how to construct authenticated bits starting from OTs. The protocol ensures that most of the authenticated bits will be kept secret, as specified by the LaBit box in Fig. 10.

The main idea of the protocol, described in Fig. 12, is the following: many authenticated bits $[y_i]_{\mathsf{B}}$ for B are created using OTs, where A is supposed to input messages $(L_i, L_i \oplus \varGamma_B)$. To check that A is using the same $\varGamma_B$ in every OT, the authenticated bits are randomly paired. Given a pair of authenticated bits $[y_i]_{\mathsf{B}}, [y_j]_{\mathsf{B}}$, A and B compute $[z_i]_{\mathsf{B}} = [y_i]_{\mathsf{B}} \oplus [y_j]_{\mathsf{B}} \oplus d_i$ where $d_i = y_i \oplus y_j$ is announced by B. If A behaved honestly, she knows the MAC that B holds on z_i, otherwise she has 1 bit of entropy on this MAC, as shown below. The parties can check if A knows the MAC using the EQ box described in Sect. 2. As B reveals $y_i \oplus y_j$, they waste $[y_j]_{\mathsf{B}}$ and only use $[y_i]_{\mathsf{B}}$ as output from the protocol—as y_j is uniformly random $y_i \oplus y_j$ leaks no information on y_i. Note that we cannot simply let A reveal the MAC on z_i, as a malicious B could announce $1 \oplus z_i$: this would allow B to learn a MAC on z_i and $1 \oplus z_i$ at the same time, thus leaking $\varGamma_B$. Using EQ forces a thus cheating B to guess the MAC on a bit which he did not see, which he can do only with negligible probability $2^{-\ell}$.

1. A samples $\varGamma_B \in_{\mathsf{R}} \{0,1\}^{\ell}$ and for $i = 1, \dots, \mathcal{T}$ samples $L_i \in_{\mathsf{R}} \{0,1\}^{\ell}$, where $\mathcal{T} = 2\tau$.
2. B samples $(y_1, \dots, y_{\mathcal{T}}) \in_{\mathsf{R}} \{0,1\}^{\mathcal{T}}$.
3. They run $\mathcal{T}$ OTs, where for $i = 1, \dots, \mathcal{T}$ party A offers $(Y_{i,0}, Y_{i,1}) = (L_i, L_i \oplus \varGamma_B)$ and B selects y_i and receives $N_i = Y_{i,y_i} = L_i \oplus y_i \varGamma_B$. Let $[y_1]_{\mathsf{B}}, \dots, [y_{\mathcal{T}}]_{\mathsf{B}}$ be the candidate authenticated bits produced so far.
4. B picks a uniformly random pairing π (a permutation $\pi : \{1, \dots, \mathcal{T}\} \to \{1, \dots, \mathcal{T}\}$ where $\forall i, \pi(\pi(i)) = i$), and sends π to A. Given a pairing π, let $\mathcal{S}(\pi) = \{i | i \leq \pi(i)\}$, i.e., for each pair, add the smallest index to $\mathcal{S}(\pi)$.
5. For all τ indices $i \in \mathcal{S}(\pi)$:
   (a) B announces $d_i = y_i \oplus y_{\pi(i)}$.
   (b) A and B compute $[z_i]_{\mathsf{B}} = [y_i]_{\mathsf{B}} \oplus [y_{\pi(i)}]_{\mathsf{B}} \oplus d_i$.
   (c) Let Z_i and W_i be the MAC and the local key for z_i held by A respectively B. They compare these using EQ and abort if they are different.

   The τ comparisons are done using *one* call on the $\tau\ell$-bit strings $(Z_i)_{i \in \mathcal{S}(\pi)}$ and $(W_i)_{i \in \mathcal{S}(\pi)}$.
6. For all $i \in \mathcal{S}(\pi)$ A and B output $[y_i]_{\mathsf{B}}$.

Fig. 12. The protocol for reducing $\mathrm{LaBit}(\tau, \ell)$ to $\mathrm{OT}(2\tau, \ell)$ and $\mathrm{EQ}(\tau\ell)$.

Note that if A uses different $\varGamma_B$ in two paired instances, $\varGamma_i$ and $\varGamma_j$ say, then the MAC held by B on $y_i \oplus y_j$ (and therefore also z_i) is $(L_i \oplus y_i \varGamma_i) \oplus (L_j \oplus y_j \varGamma_j) = (L_i \oplus L_j) \oplus (y_i \oplus y_j)\varGamma_j \oplus y_i(\varGamma_i \oplus \varGamma_j)$. Since $(\varGamma_i \oplus \varGamma_j) \neq 0^{\ell}$ and $y_i \oplus y_j$ is fixed by announcing d_i, guessing this MAC is equivalent to guessing y_i. As A only knows $L_i, L_j, \varGamma_i, \varGamma_j$ and $y_i \oplus y_j$, she cannot guess y_i with probability better than $1/2$. Therefore, if A cheats in many OTs, she will get caught with high probability. If she only cheats on a few instances she might pass the test. Doing so confirms her guess on y_i in the pairs where she cheated. Now assume that she cheated in instance i and offered $(L_i, L_i \oplus \varGamma'_B)$ instead of $(L_i, L_i \oplus \varGamma_B)$. After getting her guess on y_i confirmed she can explain the run as an honest run: If $y_i = 0$, the run is equivalent to having offered $(L_i, L_i \oplus \Gamma_B)$, as B gets no information on the second message when $y_i = 0$. If $y_i = 1$, then the run is equivalent to having offered $(L'_i, L'_i \oplus \Gamma_B)$ with $L'_i = L_i \oplus (\Gamma_B \oplus \Gamma'_B)$, as $L'_i \oplus \Gamma_B = L_i \oplus \Gamma_B$ and B gets no information on the first message when $y_i = 1$. So, any cheating strategy of A can be simulated by letting her honestly use the same Γ_B in all pairs and then let her try to guess some bits y_i. If she guesses wrong, the deviation is reported to B. If she guesses right, she is told so and the deviation is not reported to B. This, in turn, can be captured using some appropriate class of leakage functions $\mathcal{L}$. Nailing down the exact $\mathcal{L}$ needed to simulate a given behavior of A, including defining what is the "right" Γ_B, and showing that the needed $\mathcal{L}$ is always κ-secure is a relatively straight-forward but very tedious business. The proof of the following theorem can be found in the full version.

Theorem 4. *Let $\kappa = \frac{3}{4}\tau$, and let $\mathcal{L}$ be a κ secure leakage function on τ bits. The protocol in Fig. 12 securely implements $\mathrm{LaBit}^{\mathcal{L}}(\tau, \ell)$ in the $(\mathrm{OT}(2\tau, \ell), \mathrm{EQ}(\tau\ell))$-hybrid model. The communication is $O(\tau^2)$. The work is $O(\tau\ell)$.*

Corollary 1. *Let ψ denote the security parameter and let $\ell = \mathrm{poly}(\psi)$. The box $\mathrm{aBit}(\ell, \psi)$ can be reduced to $(\mathrm{OT}(\frac{44}{3}\psi, \psi), \mathrm{EQ}(\psi))$. The communication is $O(\psi\ell + \psi^2)$ and the work is $O(\psi^2\ell)$.*

Proof. Combining the above theorems we have that $\mathrm{aBit}(\ell, \psi)$ can be reduced to $(\mathrm{OT}(\frac{44}{3}\psi, \ell), \mathrm{EQ}(\frac{22}{3}\psi\ell))$ with communication $O(\psi^2)$ and work $O(\psi^2\ell)$. For any polynomial ℓ, we can implement $\mathrm{OT}(\frac{44}{3}\psi, \ell)$ given $\mathrm{OT}(\frac{44}{3}\psi, \psi)$ and a pseudo-random generator $\mathrm{prg} : \{0,1\}^{\psi} \to \{0,1\}^{\ell}$. Namely, seeds are sent using the OTs and the prg is used to one-time pad encrypt the messages. The communication is 2ℓ. If we use the RO to implement the pseudo-random generator and count the hashing of κ bits as $O(\kappa)$ work, then the work is $O(\ell\psi)$. We can implement $\mathrm{EQ}(\frac{22}{3}\psi\ell)$ by comparing short hashes produced using the RO. The work is $O(\psi\ell)$. $\square$

Since the oracles $(\mathrm{OT}(\frac{44}{3}\psi, \psi), \mathrm{EQ}(\psi))$ are independent of ℓ, the cost of essentially any reasonable implementation of them can be amortized away by picking ℓ large enough. See the full version for a more detailed complexity analysis.

Efficient OT Extension: We notice that the WaBit box resembles an intermediate step of the OT extension protocol of [18]. Completing their protocol (i.e., "hashing away" the fact that all messages pairs have the same XOR), gives an efficient protocol for OT extension, with the same asymptotic complexity as [15], but with dramatically smaller constants. See the full version for details.

5 Authenticated Oblivious Transfer

In this section we show how to implement aOTs. We implemented aBits in Sect. 4, so what remains is to show how to implement aOTs from aBits i.e., to implement the $\mathcal{F}_{\mathrm{DEAL}}$ box when it outputs

$$\begin{tikzcd}
 & \mathrm{aOT} & \\
 & \mathrm{LaOT} \arrow[u] & \\
\mathrm{aBit} \arrow[ur] & & \mathrm{EQ} \arrow[ul]
\end{tikzcd}$$

Fig. 13. Sect. 5 outline.

$[x_0]_\mathsf{A}, [x_1]_\mathsf{A}, [c]_\mathsf{B}, [z]_\mathsf{B}$ with $z = c(x_0 \oplus x_1) \oplus x_0 = x_c$.
Because of symmetry we only show the construction of aOTs from aBits with A as sender and B as receiver.

Honest Parties: For $i = 1, \dots, \ell$, the box outputs random $[x_0^i]_\mathsf{A}, [x_1^i]_\mathsf{A}, [c^i]_\mathsf{B}, [z^i]_\mathsf{B}$ with $z^i = c^i(x_0^i \oplus x_1^i) \oplus x_0^i$.

Corrupted Parties:

1. If B is corrupted he gets to choose all his random values.
2. If A is corrupted she gets to choose all her random values. Also, she may, at any point before B received his outputs, input (i, g^i) to the box in order to try to guess c^i. If $c^i \neq g^i$ the box will output `fail` and terminate. Otherwise the box proceeds as if nothing has happened and A will know the guess was correct. She may input as many guesses as she desires.

Global Key Queries: The adversary can at any point input $(\mathsf{A}, \varDelta)$ and will be returned whether $\varDelta = \varDelta_B$. And it can at any point input $(\mathsf{B}, \varDelta)$ and will be returned whether $\varDelta = \varDelta_A$.

Fig. 14. The Leaky Authenticated OT box $\mathrm{LaOT}(\ell)$

The protocol runs ℓ times in parallel. The description is for a single run of LaOT.

1. A and B get $[x_0]_\mathsf{A}, [x_1]_\mathsf{A}, [c]_\mathsf{B}$ and $[r]_\mathsf{B}$ from the dealer using the aBit box.
2. Let $[x_0]_\mathsf{A} = (x_0, M_{x_0}, K_{x_0}), [x_1]_\mathsf{A} = (x_1, M_{x_1}, K_{x_1}), [c]_\mathsf{B} = (c, M_c, K_c), [r]_\mathsf{B} = (r, M_r, K_r)$.
3. A chooses random strings $T_0, T_1 \in \{0,1\}^\kappa$.
4. A sends (X_0, X_1) to B where $X_0 = H(K_c) \oplus (x_0 || M_{x_0} || T_{x_0})$ and $X_1 = H(K_c \oplus \varDelta_B) \oplus (x_1 || M_{x_1} || T_{x_1})$.
5. B computes $(x_c || M_{x_c} || T_{x_c}) = X_c \oplus H(M_c)$. B aborts if $M_{x_c} \neq K_{x_c} \oplus x_c \varDelta_A$. Otherwise, let $z = x_c$.
6. B announces $d = z \oplus r$ to A and the parties compute $[z]_\mathsf{B} = [r]_\mathsf{B} \oplus d$. Let $[z]_\mathsf{B} = (z, M_z, K_z)$.
7. A sends (I_0, I_1) to B where $I_0 = H(K_z) \oplus T_1$ and $I_1 = H(K_z \oplus \varDelta_B) \oplus T_0$.
8. B computes $T_{1 \oplus z} = I_z \oplus H(M_z)$. *Notice that now* B *has both* (T_0, T_1).
9. A and B both input (T_0, T_1) to EQ. The comparisons are done using one call to $\mathrm{EQ}(\ell 2\kappa)$.
10. If the values are the same, they output $[x_0]_\mathsf{A}, [x_1]_\mathsf{A}, [c]_\mathsf{B}, [z]_\mathsf{B}$.

Fig. 15. The protocol for authenticated OT with leaky choice bit

We go via a leaky version of authenticated OT, or LaOT, described in Fig. 14. The LaOT box is leaky in the sense that choice bits may leak when A is corrupted: a corrupted A is allowed to make guesses on choice bits, but if the guess is wrong the box aborts revealing that A is cheating. This means that if the box does not abort, with very high probability A only tried to guess a few choice bits.

The protocol to construct a leaky aOT, described in Fig. 15, proceeds as follows: First A and B get $[x_0]_\mathsf{A}, [x_1]_\mathsf{A}$ (A's messages), $[c]_\mathsf{B}$ (B's choice bit) and $[r]_\mathsf{B}$. Then A

1. A and B generate $\ell' = B\ell$ authenticated OTs using $\mathrm{LaOT}(\ell')$. If the box does not abort, name the outputs $\{[x_0^i]_\mathsf{A}, [x_1^i]_\mathsf{A}, [c^i]_\mathsf{B}, [z^i]_\mathsf{B}\}_{i=1}^{\ell'}$.
2. B sends a B-wise independent permutation π on $\{1, \ldots, \ell'\}$ to A. For $j = 0, \ldots, \ell - 1$, the B quadruples $\{[x_0^{\pi(i)}]_\mathsf{A}, [x_1^{\pi(i)}]_\mathsf{A}, [c^{\pi(i)}]_\mathsf{B}, [z^{\pi(i)}]_\mathsf{B}\}_{i=jB+1}^{jB+B}$ are defined to be in the j-th bucket.
3. We describe how to combine two OTs from a bucket, call them $[x_0^1]_\mathsf{A}, [x_1^1]_\mathsf{A}, [c^1]_\mathsf{B}, [z^1]_\mathsf{B}$ and $[x_0^2]_\mathsf{A}, [x_1^2]_\mathsf{A}, [c^2]_\mathsf{B}, [z^2]_\mathsf{B}$. Call the result $[x_0]_\mathsf{A}, [x_1]_\mathsf{A}, [c]_\mathsf{B}, [z]_\mathsf{B}$. To combine more than two, just iterate by taking the result and combine it with the next leaky OT.
   (a) A reveals $d = x_0^1 \oplus x_1^1 \oplus x_0^2 \oplus x_1^2$.
   (b) Compute: $[c]_\mathsf{B} = [c^1]_\mathsf{B} \oplus [c^2]_\mathsf{B}$, $[z]_\mathsf{B} = [z^1]_\mathsf{B} \oplus [z^2]_\mathsf{B} \oplus d[c^1]_\mathsf{B}$, $[x_0]_\mathsf{A} = [x_0^1]_\mathsf{A} \oplus [x_0^2]_\mathsf{A}$, $[x_1]_\mathsf{A} = [x_0^1]_\mathsf{A} \oplus [x_1^2]_\mathsf{A}$.

Fig. 16. From Leaky Authenticated OTs to Authenticated OTs

transfers the message $z = x_c$ to B in the following way: B knows the MAC for his choice bit M_c, while A knows the two keys K_c and Δ_B. This allows A to compute the two possible MACs $(K_c, K_c \oplus \Delta_B)$ respectively for the case of $c = 0$ and $c = 1$. Hashing these values leaves A with two uncorrelated strings $H(K_c)$ and $H(K_c \oplus \Delta_B)$, one of which B can compute as $H(M_c)$. These values can be used as a one-time pad for A's bits x_0, x_1 (and some other values as described later). B can retrieve x_c and announce the difference $d = x_c \oplus r$ and therefore compute the output $[z]_\mathsf{B} = [r]_\mathsf{B} \oplus d$.

In order to check if A is transmitting the correct bits x_0, x_1, she will transfer the respective MACs together with the bits: as B is supposed to learn x_c, revealing the MAC on this bit does not introduce any insecurity. However, A can now mount a selective failure attack: A can check if B's choice bit c is equal to, e.g., 0 by sending x_0 with the right MAC and x_1 together with a random string. Now if $c = 0$ B only sees the valid MAC and continues the protocol, while if $c = 1$ B aborts because of the wrong MAC. A similar attack can be mounted to check if $c = 1$. We will fix this later by randomly partitioning and combining a few LaOTs together.

On the other hand, if B is corrupted, he could be announcing the wrong value d. In particular, A needs to check that the authenticated bit $[z]_\mathsf{B}$ is equal to x_c without learning c. In order to do this, we have A choosing two random strings T_0, T_1, and append them, respectively, to x_0, x_1 and the MACs on those bits, so that B learns T_c together with x_c. After B announces d, we can again use the MAC and the keys for z to perform a new transfer: A uses $H(K_z)$ as a one-time pad for T_1 and $H(K_z \oplus \Delta_B)$ as a one-time pad for T_0. Using M_z, the MAC on z, B can retrieve $T_{1 \oplus z}$. This means that an honest B, that sets $z = x_c$, will know both T_0 and T_1, while a dishonest B will not be able to know both values except with negligible probability. Using the EQ box A can check that B knows both values T_0, T_1. Note that we cannot simply have B openly announce these values, as this would open the possibility for new attacks on A's side. The proof of the following theorem can be found in the full version.

Theorem 5. *The protocol in Fig. 15 securely implements* $\mathrm{LaOT}(\ell)$ *in the* $(\mathrm{aBit}(4\ell, \kappa), \mathrm{EQ}(2\ell\kappa))$*-hybrid model.*

To deal with the leakage of the LaOT box, we let B randomly partition the LaOTs in small buckets: all the LaOTs in a bucket will be combined using an OT combiner, as shown in Fig. 16, in such a way that if at least one choice bit in every bucket is unknown to A, then the resulting aOT will not be leaky. The overall protocol is secure because of the OT combiner and the probability that any bucket is filled only with OTs where the choice bit leaked is negligible, as shown in the full version.

Theorem 6. *Let* $\mathrm{aOT}(\ell)$ *denote the box which outputs* ℓ aOT*s as in* $\mathcal{F}_{\mathrm{DEAL}}$. *If* $(\log_2(\ell)+1)(B-1) \geq \psi$*, then the protocol in Fig. 16 securely implements* $\mathrm{aOT}(\ell)$ *in the* $\mathrm{LaOT}(B\ell)$*-hybrid model with security parameter* ψ.

6 Experimental Results

We did a proof-of-concept implementation in Java. The hash function in our protocol was implemented using Java's standard implementation of SHA256. The implementation consists of a circuit-independent protocol for preprocessing all the random values output by $\mathcal{F}_{\mathrm{DEAL}}$, a framework for constructing circuits for a given computation, and a run-time system which takes preprocessed values, circuits and inputs and carry out the secure computation.

We will not dwell on the details of the implementation, except for one detail regarding the generation of the circuits. In our implementation, we do not compile the function to be evaluated into a circuit in a separate step. The reason is that this would involve storing a huge, often highly redundant, circuit on the disk, and reading it back. This heavy disk access turned out to constitute a significant part of the running time in an earlier of our prototype implementations, which we discarded. Instead, in the current prototype, circuits are generated on the fly, in chunks which are large enough that their evaluation generate large enough network packages that we can amortize away communication latency, but small enough that the circuit chunks can be kept in memory during their evaluation. A circuit compiled is hence replaced by a succinct program which generates the circuit in a streaming manner. This circuit stream is then sent through the runtime machine, which receives a separate stream of preprocessed $\mathcal{F}_{\mathrm{DEAL}}$-values from the disk and then evaluates the circuit chunk by chunk in concert with the runtime machine at the other party in the protocol. The stream of preprocessed $\mathcal{F}_{\mathrm{DEAL}}$-values from the disk is still expensive, but we currently see no way to avoid this disk access, as the random nature of the preprocessed values seems to rule out a succinct representation.

For timing we did oblivious ECB-AES encryption. (Both parties input a secret 128-bit key K_A respectively K_B, defining an AES key $K = K_A \oplus K_B$. A inputs a secret ℓ-block message $(m_1, \ldots, m_\ell) \in \{0,1\}^{128\ell}$. B learns $(E_K(m_1), \ldots, E_K(m_\ell))$.) We used a modified version of the AES circuit from [31] and we thank Benny Pinkas, Thomas Schneider, Nigel P. Smart and Stephen C. Williams for providing us with this circuit.

The reason for using AES is that it provides a reasonable sized circuit which is also reasonably complex in terms of the structure of the circuit and the depth, as opposed to just running a lot of AND gates in parallel. Also, AES has been used for benchmark in previous implementations, like [31], which allows us to do a crude comparison to previous implementations. The comparison can only become crude, as the experiments were run in different experimental setups.

ℓ	G	σ	T_{pre}	T_{onl}	T_{tot}/ℓ	G/T_{tot}
1	34,520	55	38	4	44	822
27	922,056	55	38	5	1.6	21,545
54	1,842,728	58	79	6	1.6	21,623
81	2,765,400	60	126	10	1.7	20,405
108	3,721,208	61	170	12	1.7	20,541
135	4,642,880	62	210	15	1.7	20,637
256	8,739,200	65	406	16	1.7	20,709
512	17,478,016	68	907	26	1.8	18,733
1,024	34,955,648	71	2,303	52	2.3	14,843
2,048	69,910,912	74	5,324	143	2.7	12,788
4,096	139,821,440	77	11,238	194	2.8	12,231
8,192	279,642,496	80	22,720	258	2.8	12,170
16,384	559,284,608	83	46,584	517	2.9	11,874

Fig. 17. Timings: The reported time for to $\ell \leq 135$ is the average over 5 runs. For $\ell \geq 256$ is for single runs. Units are as follows: ℓ is number of 128-bit blocks encrypted, G is Boolean gates, σ is bits of security, $T_{\text{pre}}, T_{\text{onl}}, T_{\text{tot}}$ are seconds.

In the timings we ran A and B on two different machines on Aarhus University's intranet (using two Intel Xeon E3430 2.40GHz cores on each machine). We recorded the number of Boolean gates evaluated (G), the time spent in preprocessing (T_{pre}) and the time spent by the run-time system (T_{onl}). In the table in Fig. 17 we also give the amortized time per AES encryption (T_{tot}/ℓ with $T_{\text{tot}} \overset{\text{def}}{=} T_{\text{pre}} + T_{\text{onl}}$) and the number of gates handled per second (G/T_{tot}). The time T_{pre} covers the time spent on computing and communicating during the generation of the values preprocessed by $\mathcal{F}_{\text{DEAL}}$, and the time spent storing these value to a local disk. The time T_{onl} covers the time spent on generating the circuit and the computation and communication involved in evaluating the circuit given the values preprocessed by $\mathcal{F}_{\text{DEAL}}$.

We work with two security parameters. The computational security parameter κ specifies that a poly-time adversary should have probability at most $\text{poly}(\kappa)2^{-\kappa}$ in breaking the protocol. The statistical security parameter σ specifies that we allow the protocol to break with probability $2^{-\sigma}$ independent of the computational power of the adversary. As an example of the use of κ, our keys and therefore MACs have length κ. This is needed as the adversary learns $H(K_i)$ and $H(K_i \oplus \Delta)$ in our protocols and can break the protocol given Δ. As an example of the use of σ, when we generate ℓ gates with bucket size B, then $\sigma \leq (\log_2(\ell)+1)(B-1)$ due to the probability $(2\ell)^{1-B}$ that a bucket might end up containing only leaky components. This probability is independent of the computational power of the adversary, as the components are being bucketed by the honest party after it is determined which of them are leaky.

In the timings, the computational security parameter has been set to 120. Since our implementation has a fixed bucket size of 4, the statistical security level depends on ℓ. In the table, we specify the statistical security level attained (σ means insecurity $2^{-\sigma}$). At computational security level 120, the implementation needs to do 640 seed OTs. The timings do not include the time needed to do these, as that would depend on the implementation of the seed OTs, which is not the focus here. We note, however, that using, e.g., the implementation in [31], the seed OTs could be done in around 20 seconds, so they would not significantly affect the amortized times reported.

The dramatic drop in amortized time from $\ell = 1$ to $\ell = 27$ is due to the fact that the preprocessor, due to implementation choices, has a smallest unit of gates it can preprocess for. The largest number of AES circuits needing only one, two, three, four and five units is 27, 54, 81, 108 and 135, respectively. Hence we preprocess equally many gates when $\ell = 1$ and $\ell = 27$.

As for total time, we found the best amortized behavior at $\ell = 54$, where oblivious AES encryption of one block takes amortized 1.6 seconds, and we handle 21,623 gates per second. As for online time, we found the best amortized behavior at $\ell = 2048$, where handling one AES block online takes amortized 32 milliseconds, and online we handle 1,083,885 gates per second. We find these timings encouraging and we plan an implementation in a more machine-near language, exploiting some of the findings from implementing the prototype.

References

1. a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT*, pages 386–405, 2011.
2. B. Applebaum, D. Harnik, and Y. Ishai. Semantic security under related-key attacks and applications. In *ICS*, pages 45–60, 2011.
3. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS*, pages 257–266, 2008.
4. R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.
5. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
6. S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *CT-RSA*, pages 416–432, 2012.
7. S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-xor" technique. In *TCC*, pages 39–53, 2012.
8. C. Crépeau, J. van de Graaf, and A. Tapp. Committed oblivious transfer and private multi-party computation. In *CRYPTO*, pages 110–123, 1995.
9. I. Damgård and M. Keller. Secure multiparty aes. In *Financial Cryptography*, pages 367–374, 2010.
10. I. Damgård and C. Orlandi. Multiparty computation for dishonest majority: From passive to active security at low cost. In *CRYPTO*, pages 558–576, 2010.
11. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324, 2005.
12. J. A. Garay, P. MacKenzie, and K. Yang. Efficient and universally composable committed oblivious transfer and applications. In *TCC*, pages 297–316, 2004.
13. O. Goldreich. *Foundations of Cryptography, Vol. 2*. Cambridge University Press, 2004. `http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html`.
14. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.
15. D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-combiners via secure computation. In *TCC*, pages 393–411, 2008.

16. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS*, pages 451–462, 2010.
17. Y. Huang, D. Evans, J. Katz, , and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.
18. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO*, pages 145–161, 2003.
19. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.
20. Y. Ishai, M. Prabhakaran, and A. Sahai. Founding cryptography on oblivious transfer - efficiently. In *CRYPTO*, pages 572–591, 2008.
21. Y. Ishai, M. Prabhakaran, and A. Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.
22. T. P. Jakobsen, M. X. Makkes, and J. D. Nielsen. Efficient implementation of the orlandi protocol. In *ACNS*, pages 255–272, 2010.
23. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *ICALP (2)*, pages 486–498, 2008.
24. Y. Lindell, E. Oxman, and B. Pinkas. The ips compiler: Optimizations, variants and concrete efficiency. In *CRYPTO*, pages 259–276, 2011.
25. Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. TCC, 2011.
26. Y. Lindell, B. Pinkas, and N. P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *SCN*, pages 2–20, 2008.
27. L. Malka and J. Katz. VMCrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584, 2010. `http://eprint.iacr.org/`.
28. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.
29. J. B. Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. Cryptology ePrint Archive, Report 2007/215, 2007. `http://eprint.iacr.org/`.
30. J. B. Nielsen and C. Orlandi. Lego for two-party secure computation. In *TCC*, pages 368–386, 2009.
31. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*, pages 250–267, 2009.
32. A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.