New Generation Computing (2022) 40:115–147
https://doi.org/10.1007/s00354-021-00149-9

Private Function Evaluation with Cards

Alexander Koch[1] · Stefan Walzer[2]

Received: 2 September 2021 / Accepted: 15 December 2021 / Published online: 12 February 2022

Abstract

Card-based protocols allow to evaluate an arbitrary fixed Boolean function f on a hidden input to obtain a hidden output, without the executer learning anything about either of the two (e.g., [12]). We explore the case where f implements a universal function, i.e., f is given the encoding $\langle P\rangle$ of a program P and an input x and computes $f(\langle P\rangle, x) = P(x)$. More concretely, we consider universal circuits, Turing machines, RAM machines, and branching programs, giving secure and conceptually simple card-based protocols in each case. We argue that card-based cryptography can be performed in a setting that is only very weakly interactive, which we call the "surveillance" model. Here, when Alice executes a protocol on the cards, the only task of Bob is to watch that Alice does not illegitimately turn over cards and that she shuffles in a way that nobody knows anything about the total permutation applied to the cards. We believe that because of this very limited interaction, our results can be called *program obfuscation*. As a tool, we develop a useful sub-protocol $\mathsf{sort}_{\Pi}X{\uparrow}Y$ that couples the two equal-length sequences X, Y and jointly and obliviously permutes them with the permutation $\pi \in \Pi$ that lexicographically minimizes $\pi(X)$. We argue that this generalizes ideas present in many existing card-based protocols. In fact, AND, XOR, bit copy [37], coupled rotation shuffles [30] and the "permutation division" protocol of [22] can all be expressed as "coupled sort protocols".

Keywords Card-based protocols · RAM machine · Branching program · Secure computation · Universal circuits · Obfuscation · Cryptography without computers

The content of this paper appeared in the first author's dissertation [26].

✉ Alexander Koch
alexander.koch@kit.edu

Stefan Walzer
walzer@cs.uni-koeln.de

[1] Competence Center for Applied Security Technology (KASTEL), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

[2] University of Cologne, Cologne, Germany

Introduction

Secure multiparty computation (MPC) allows multiple players to jointly compute a function, without giving away anything about their inputs, except what can be deduced from the output. An important special case is when the function to be evaluated constitutes an input itself and should remain hidden, called *Private Function Evaluation* (PFE). This has been considered in the standard cryptographic setting, e.g., using universal circuits [45] in [5, 19, 32, 38].

Secure multiparty computation, and hence also PFE (by choosing a universal function to be executed), can also be done with a *deck of physical cards*, as first shown in [8, 12, 41]. In this area of *card-based cryptography*, one designs tangible protocols using a deck of cards with information-theoretic privacy features. There is already a wealth of literature on how to jointly and securely compute an arbitrary (fixed) circuit on the players' inputs, see, e.g., [12, 37, 41]. Moreover, similar but different physical assumptions have been exploited in other settings, in particular in the cryptographic voting community, cf. Scantegrity, PunchScan, and Oblivious voting [2, 10, 11, 43] (see [28] for a survey on physical assumptions in cryptography).

Motivation. Card-based protocols are often used in educational and recreational settings. For an illustration of PFE, we stretch the usual motivation for card-based AND protocols a bit, namely the dating problem where players want to find out whether there is mutual love.

We assume a predefined set of binary attributes A such as $A = \{\texttt{LikesCats}, \texttt{HasPhD}, \texttt{IsGeeky},\ldots\}$. Alice implicitly specifies (by providing a circuit or program) which combinations $P \subseteq 2^A$ of attributes she likes and Bob specifies which attributes $B \subseteq A$ he has. The task is to determine whether Bob's secret attributes satisfy Alice's secret preferences, i.e., whether $B \in P$. Here, we want to ensure that both Alice's and Bob's input remains hidden, i.e., nothing about the input is revealed, except what can be deduced from the output of the protocol.

In the same vein, PFE is useful for the game *Skipjack* [16][1], where a game master invents a rule and the other players take turns querying whether a chosen code words satisfies the rule or not—to deduce/guess the rule in this process. Applying our PFE protocol would allow to prevent the game master from cheating by changing the rule mid-game, or even to play the game in absence of a game master, assuming an encoding of a rule is available or can be obtained at random. (Moreover, as PFE even hides the code words that the player is testing, we can derive a competitive multi-player mode where questions of other players do not help the others.)

Look and Feel of Our Protocols. Imagine a room with a table, where Alice puts an encoding of a function f in a sequence on the table, each bit of the description as two face-down cards encoding 0 via ♣,♡ and 1 via ♡ ♣. Next to Alice's cards, Bob will put his input x as a bit string using the same encoding. The game then proceeds according to a protocol (described in more detail later) that may prescribe to (i) shuffle the cards in certain controlled ways and (ii) turn over cards (the observed

[1] a follow-up on a game by Abbott [1] from 1956. Skipjack was given as a present to all participants of ASIACRYPT 2015.

symbols may affect the future course of the protocol). The protocol terminates with output $f(x)$ encoded as face-down cards. The output can then be revealed to both players or used obliviously in further computations.

The Sort Sub-protocol. The protocols proposed in this paper—and actually a large subset of the protocols from the literature—can be regarded as a sequence of sub-protocols with basically the same functionality, which we capture under the name "sort protocol". We believe this observation is of independent interest. We also show that, under weak assumptions, protocols obtained as compositions of sort-protocols are secure. This elegantly re-proves the security of existing protocols and greatly simplifies the security proofs of our own protocols. (As we are in a simpler and fully information-theoretic setting, this is much easier than in the common universal composability framework [9]).

On Interaction in Card-Based Protocols. We point out that card-based cryptography can be assumed secure in a rather *non-interactive* physical model: it suffices to have one protocol executer, who is under surveillance by the other players. For example, when the protocol description specifies that a certain shuffle is to be performed, this step can be implemented by this one player, the executer, who uses envelopes (or helping cards) and completely random shuffles or uniform random cuts in a manner that ensures that not even he himself can keep track of concrete permutation done on the cards. (We could also use shuffling machines, such as the wheel-of-fortune-esque device in [46].)

Note that in this *surveillance model* where players watch that the protocol is done correctly, many protocols can be argued secure with almost no interaction. For example, ([21], Protocol 3) is a nice physical zero-knowledge proof system for proving that there is a solution to a Sudoku puzzle, where the verifier chooses one of three cards in each cells of the Sudoku to be assigned to piles for rows, columns and subgrids to be able to later verify that all numbers are present. In our model, we can plausibly argue that the randomness chosen by the verifier can also be directly generated by the prover himself on an additional deck of helping cards. If he is watched to perform the shuffle in a way that generates high entropy not under his control, he can use this generated randomness to assign the cards to the piles. This is actually a general observation regarding protocols using public coins, where this shuffling produces an output that can be interpreted to be like the Random Oracle output in the Fiat–Shamir heuristic. The possibility of secure shuffling in this way is a common assumption that people make when playing card games with others.

Using the PFE protocols introduced in this paper, this immediately leads to a direct way to obtain cryptographic *obfuscation* in this card-based surveillance model: assuming that the encoded protocol is lying on the table using cards, the executer can add cards encoding the inputs and then execute a universal protocol, such as the ones proposed in this paper, with *the only interaction* being guards that watch out for publicly observable deviations from the protocol.

However, note that because of the very different setting, there are no implications for the usual non-physical (strictly non-interactive) cryptographic world, where general (virtual black-box) obfuscation is impossible, cf. [7].

Universal Protocols and Their Qualities. We implement four different universal card-based protocols with varying degrees of abstraction, based on branching programs, circuits, Turing machines and RAM machines. Our primary focus is on simplicity and elegance of the protocols, but we also consider efficiency in terms of runtime and required cards.

The benefit of providing several solutions is that depending on the nature of the task, a certain computational model may be particularly suitable. For example, in the generalized dating game described above, using universal circuits is a natural option, while a rule in Skipjack might most naturally be described as a program using loops and thus benefit from the possibilities available in Turing machines and RAM machines. For didactic settings, all options are interesting in itself, as they demonstrate the computational models and the implemented privacy properties in a palpable way.

Contribution.

- We show how to encode and execute circuits, Turing machines, RAM machines and branching programs with cards and specify protocols for executing these on hidden inputs so that nothing about the machine description (except the length, etc.) or the inputs is leaked. We achieve this using envelopes and only very natural shuffle operations, namely random cuts and S_n-shuffles (i.e., ordinary shuffling, where all card reorderings are equally likely).
- Given the weakly interactive nature of card-based cryptography in the "surveillance model" (see above), we thereby obtain what may be called cryptographic obfuscation in a card-based setting.
- We identify and generalize a primitive that is the basis for many protocols and operations in cards-based cryptography, namely *coupled sorting*, cf. Sect. 3.

Related Work. Regarding our branching program construction, let us mention that there are several card-based protocols to randomly generate a permutation with specific, prescribed properties. For example, the secret santa game asks for random permutations on the player indices (encoding who gives a present to whom) that are fixed-point free to ensure that nobody receives their own present, and has been implemented with cards in [12, 23]. Moreover, they also give protocols for generating permutations with cycles of a certain minimal length. Moreover, Hashimoto et al. [22] give a protocol for generating permutations with a prespecified cycle structure, and show how to obliviously execute the inverse of a permutation encoded with cards on another card sequence, which is a special case of our sorting operations. In general, we make use of card decks that not only feature heart or clubs cards, a line of research that was pursued in, e.g., [29, 35, 42].

Note that cryptographic obfuscation has been performed in other models. For example, Goyal et al. [18] make use of tamper-proof hardware tokens (such as smart cards) introduced by Katz [24]. Moreover, [36] allows to execute many cryptographic primitives (albeit not obfuscation) using scratch-off cards. They have a slightly weaker setting, as they do not gather players around a table, but use sealed (tamper-evident) envelopes that are sent between the players via mail, getting out-of-sight from the other players.

Physical computation is also described in [13] (as "Physical GMW protocol") to achieve security in the framework of Universal Composability with Local

Adversaries (LUC). However, they make very strong assumptions on available “machines”, which we do not need.

Crépeau and Kilian [12] also discuss playing games against a card-encoded (probabilistic) circuit opponent. However, they do not aim to hide this circuit to the player as it is given by the player himself.

Recently, Dvorák and Koucký [14] formulate a similar mechanism to execute Turing machines and branching programs using cards to classify a certain class of card-based protocols that compute functions that are specified by their complexity. This constitutes independent and concurrent work.

Outline. Section 2 gives the necessary preliminaries, including the computational model used in card-based cryptography. Section 3 introduces sorting protocols as a main and versatile building block in card-based cryptography and interprets many results in the field as a single application of such a protocol. We describe concrete protocols for executing universal circuits (Sect. 4), Turing machines (Sect. 5), (word-)RAM machines (Sect. 6) and branching programs (Sect. 7).

Notation (Permutations). For distinct elements $x_1, \dots, x_k \in X$ the *cycle* $(x_1\ x_2\ \dots\ x_k)$ denotes the *cyclic* permutation π with $\pi(x_i) = x_{i+1}$ for $1 \leq i < k$, $\pi(x_k) = x_1$, and $\pi(x) = x$ for all $x \in X$ not occurring in the cycle. For multiple cycles on pairwise disjoint sets, we write them next to one another to denote their composition, e.g., $(1\ 2)(3\ 4\ 5)$ maps $1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 4, 4 \mapsto 5, 5 \mapsto 3$.

Computational Model of Card-Based Cryptography

Card-based protocols operate on a *deck* of cards, which is specified by a multiset $\mathcal{D}$ of symbols, e.g., from $\{\heartsuit, \clubsuit\}$ or from numbered cards $\{1, \dots, n\}$. It uses four operations, namely *i)* turning over cards to reveal their hidden symbols, *ii)* deterministically permuting the cards, *iii)* shuffling the cards in some controlled way to introduce randomness, and *iv)* terminating and outputting a list of card positions encoding the protocol output. The formal model is given in [39].

While many protocols in the literature only use $\{\heartsuit, \clubsuit\}$ as a deck alphabet, Niemi and Renvall [42] and Mizuki [35] introduce card-based protocols using the (multi-)set $[1, \dots, n]$, and an encoding rule, where a bit given by two face-down cards is 0 if the former card has a smaller value, and 1 otherwise.

More formally, a *protocol* $\mathcal{P}$ is a quadruple $(\mathcal{D}, U, Q, A)$, where $\mathcal{D}$ is a deck, U is a set of input sequences over $\mathcal{D}$, Q is a set of states with $q_0 \in Q$ and $q_{\text{fin}} \in Q$, being the initial and the final state. Moreover, we have an action function $A : (Q \setminus \{q_{\text{fin}}\}) \times \mathsf{Vis}^{\mathcal{D}} \to Q \times \mathsf{Action}$, depending on the current state and visible sequence (i.e., the sequence of the card symbols, with face-down cards specified as a special back symbol ‘?’, and face-up cards showing their symbol; the set of visible sequences on deck $\mathcal{D}$ is denoted by $\mathsf{Vis}^{\mathcal{D}}$), which specifies the next state and an operation on the sequence. These actions, constituting the set Action are as follows, performed on a sequence $\Gamma = (\Gamma[1], \dots, \Gamma[n])$:

i) (turn, T), for a set $T \subseteq \{1, \dots, n\}$, flips the cards at positions specified by the *turn set* T. Formally, for a card $c = \frac{a}{b}$ we define $\mathsf{swap}(c) := \frac{b}{a}$ and transform Γ into $\mathsf{turn}_T(\Gamma)$, where $\mathsf{turn}_T(\Gamma)[i] := \mathsf{swap}(\Gamma[i])$ if $i \in T$, and $\mathsf{turn}_T(\Gamma)[i] := \Gamma[i]$, otherwise.

ii) (perm, π), for a permutation $\pi \in S_n$, permutes Γ according to π, i.e., it yields the sequence $\pi(\Gamma) = (\Gamma[\pi^{-1}(1)], \dots, \Gamma[\pi^{-1}(n)])$.

iii) $(\mathsf{shuffle}, \Pi)$, for a permutation set $\Pi \subseteq S_n$, draws a permutation $\pi \in \Pi$ uniformly at random and obliviously applies it to Γ.

iv) $(\mathsf{result}, p_1, \dots, p_r)$, for a list of distinct positions $p_1, \dots, p_r \in \{1, \dots, n\}$, halts the protocol and specifies $O = (\Gamma[p_1], \dots, \Gamma[p_r])$ as the *output*.

See [26, 39] for more details. Then, a *sequence trace* of a finite protocol run is a list $(\Gamma_0, \Gamma_1, \dots, \Gamma_t)$ of sequences such that $\Gamma_0 \in U$ and Γ_{i+1} arises from Γ_i by the specified action. Moreover, mapping this to a trace where not the cards themselves, but only what is visible about the cards, is called the corresponding *visible sequence trace*.

Card-based protocols are secure if input and output are perfectly hidden, i.e., from the outside the execution of a protocol has the same distribution, regardless of what input and output are.

Definition 2.1 (Security, cf. [30, 31][2]) Let $\mathcal{P} = (\mathcal{D}, U, Q, A)$ be a protocol. It is *(input- and output-)secure* if for any random variable I with values in the set of input sequences U, the following holds. A protocol run starting with random initial sequence $\Gamma_0 = I$, and taking random choices for the shuffling actions, terminates almost surely (i.e., with probability 1). Further, if V and O are random variables denoting the visible sequence trace and the output of the run, then the pair (I, O) is stochastically independent of V.

Boolean Circuits A *Boolean circuit* with l input variables $v_1, \dots, v_l$ is a directed acyclic graph $C = (V, E)$. The nodes are called gates and are labeled with $\vee$, $\wedge$, $\neg$, an input variable, or one of the constants 1 or 0. In the cases of $\vee$, $\wedge$, $\neg$, the in-degree must be 2, 2 or 1, respectively, otherwise it is 0. The *output node* is the unique node with out-degree 0. The *depth* of C is the maximum number of $\wedge$ and $\vee$ gates on a path in C.

The value $C(\vec{v}) \in \{0, 1\}$ that a circuit outputs on input $\vec{v} = (v_1, \dots, v_l) \in \{0, 1\}^l$ is defined in the natural way. For this paper, it is convenient to transform all $\vee$-gates into $\wedge$-gates using de Morgan's rule $(x \vee y) = \neg(\neg x \wedge \neg y)$. Note that this transformation does not affect the depth of the circuit.

Group Actions. In Sect. 3, we make use of group actions and their orbits, which can be found, e.g., in ([15], Sect. 1.3). For a definition, let X be a nonempty set, G a group, and $\varphi : G \times X \to X$ a function implicit in the notation $g(x) := \varphi(g, x)$ for $g \in G, x \in X$. G *acts* on X, or φ is a *group action* on X if

[2] Note that this notion also captures security against players that have partial information about input and output, cf. ([30], Sect. 6 (Delegated Computation)).

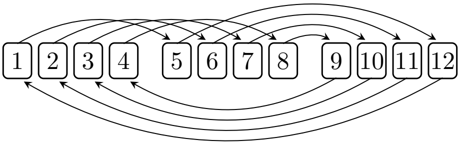


Fig. 1 Effect of the permutation $(1\ 2\ 3)\uparrow((1,2,3,4),(5,6,7,8),(12,11,10,9))$ when applied to a sequence $(1,\dots,12)$ of cards. The idea is to permute the three card sequences in positions (1, 2, 3, 4), (5, 6, 7, 8) and (12, 11, 10, 9) (all of same length) cyclically (as in (1 2 3)), taking the groups of four cards "as a whole". To illustrate the possibility of given the sequences in the operation in another order, we reversed the third sequence with the effect that when (5, 6, 7, 8) is "mapped" to (12, 11, 10, 9), the card at the 5th position is mapped to the 12th position, and so on (as displayed in the figure)

- $\mathsf{id}(x) = x$ for all $x \in X$, where id denotes the neutral element in G,
- $(g \circ h)(x) = g(h(x))$ for all $x \in X$ and all $g, h \in G$.

Let G be a group acting on a set X. Then, the *orbit* of an $x \in X$ is $G(x) := \{g(x) : g \in G\}$, i.e., all elements in X that are reachable from x via some $g \in G$. Note that orbits $G(x), G(y)$ of $x, y \in X$ are either disjoint or equal. Hence the orbits form a partition of X, called the *orbit partition* of X through G. For an application of this to proving lower bounds on the number of cards in card protocols, see [25]. In our setting, $G = \Pi \subseteq S_n$ is a permutation subgroup used in a shuffle and X is the set of sequences over a deck $\mathcal{D}$. Then, Π acts on X by permuting the card sequences $x \in X$ via $\pi \in \Pi$, i.e., $\pi((x_1,\dots,x_n)) = (x_{\pi^{-1}(1)},\dots,x_{\pi^{-1}(n)})$.

The Coupled Sorting Sub-protocol

In this section, we introduce our main, versatile building block, namely "sorting protocols", and later show how to interpret many protocols from the literature as such a protocol. We use the term "coupled" to indicate that a same permutation is applied to multiple card subsequences by forming piles (e.g., to be placed in envelopes) and then permuting them, cf. Fig. 2.

Notation. Let $\pi \in S_n$, $A = (a_1,\dots,a_n)$ a sequence of distinct natural numbers and B a sequence of length n. We define the *lift* $\pi\uparrow A$ of π to A via

$$(\pi\uparrow A)(m) := \begin{cases} a_{\pi(i)}, & \text{if } m = a_i \text{ for some } i, \\ m, & \text{otherwise,} \end{cases}$$

for m with $1 \le m \le \max\{a_1,\dots,a_n\}$. For instance, the permutation $\pi = (1\ 3)(2\ 4) \in S_4$ lifted to the sequence $A = (5,2,7,8)$ yields the permutation $\pi\uparrow A = (5\ 7)(2\ 8)$. We define the *lift* of a permutation to a *sequence of same-length sequences* $B = ((b_1^1,\dots,b_1^k),\dots,(b_n^1,\dots,b_n^k))$ as

$$\pi\uparrow B := (\pi\uparrow(b_1^1,\dots,b_n^1)) \circ \cdots \circ (\pi\uparrow(b_1^k,\dots,b_n^k)).$$

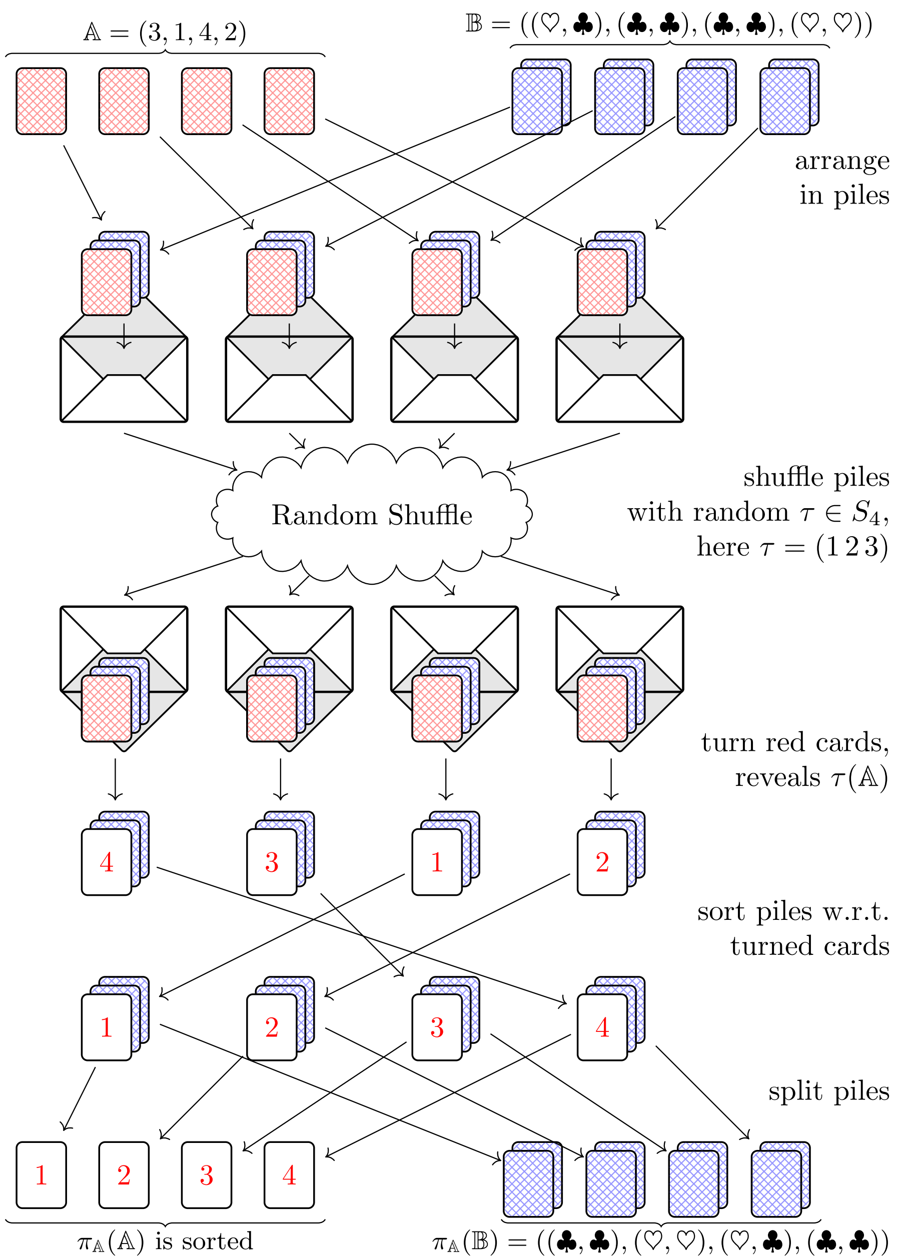

$\mathbb{A} = (3, 1, 4, 2)$
$\mathbb{B} = ((\heartsuit, \clubsuit), (\clubsuit, \clubsuit), (\clubsuit, \clubsuit), (\heartsuit, \heartsuit))$
arrange in piles
Random Shuffle
shuffle piles with random $\tau \in S_4$, here $\tau = (1\,2\,3)$
turn red cards, reveals $\tau(\mathbb{A})$
4
3
1
2
sort piles w.r.t. turned cards
1
2
3
4
split piles
1
2
3
4
$\pi_{\mathbb{A}}(\mathbb{A})$ is sorted
$\pi_{\mathbb{A}}(\mathbb{B}) = ((\clubsuit, \clubsuit), (\heartsuit, \heartsuit), (\heartsuit, \clubsuit), (\clubsuit, \clubsuit))$

Fig. 2 Application of $\mathsf{sort}_{S_4} A{\uparrow}B$ where A denotes the four positions of the red cards and B the four pairs of positions of the blue cards, in canonical ordering. Since the current sequence is $\mathbb{A} = (3, 1, 4, 2)$, the permutation $\pi_{(3,1,4,2)} = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 4, 4 \mapsto 2\} = (1\ 3\ 4\ 2)$ is applied to A and B, leaving the red cards sorted and the pairs of blue cards permuted by $\pi_{(3,1,4,2)}$ as shown. Note that the encoding of the permutation through card sequences is as in Sect. 3.2, and that the revealed sequence (4, 3, 1, 2) is independent of the input sequences and the output sequence. (The different back colors are for illustration and to avoid errors in handling the cards, but are not necessary in theory.)

Note that for each $i \in \{1, \dots, k\}$, the $(b^i_1, \dots, b^i_n)$ are again assumed to be distinct. We permit that the b^j_i are sequences again. In this sense, this definition is recursive. Figure 1 illustrates the simple intuition behind these more complex lifts.

We naturally extend this definition to permutation sets $\Pi \subseteq S_n$ and, for convenience, a lift to two sequences A, B as

$$\Pi{\uparrow}A := \{\pi{\uparrow}A \, : \, \pi \in \Pi\}, \quad \Pi{\uparrow}A, B := \{(\pi{\uparrow}A) \circ (\pi{\uparrow}B) \, : \, \pi \in \Pi\}.$$

The Family of Sort (Sub-)protocols. For each combination of a group of permutations $\Pi \subseteq S_n$, a sequence of (card) positions $A = (a_1, \dots, a_n)$ and another sequence $B = (b_1, \dots, b_n)$, we will define a "protocol" $\mathsf{sort}_\Pi A{\uparrow}B$. However, to avoid a larger and unnecessary technical exposition of sequential compositions of card-based protocols, we will use the symbol $\mathsf{sort}_\Pi A{\uparrow}B$ just as a shorthand or syntactic sugar for the sequence of four actions as stated in Protocol 1 (which is explained below). As this behaves like an inlined function in programming languages, we chose to call it "(sub-)protocol" in the following.

Note that Π, A and B are a public part of the action specification, not inputs. To describe the intended behavior of the shorthand, assume it is executed on a sequence Γ of cards. Let $\mathbb{A} := \Gamma[A] := (\Gamma[a_1], \dots, \Gamma[a_n])$ be the sequence of cards in positions A, and $\mathbb{B} := \Gamma[B]$ the sequence of cards in positions B. We assume that these card (symbol) sequences $\mathbb{A}$ and $\mathbb{B}$ are secret.

Protocol 1. $\mathsf{sort}_\Pi\ A \uparrow B$ (using a deck compatible with A and B):

```
$(\mathsf{shuffle}, \Pi \uparrow A, B)$ // chooses $\tau \in \Pi$ randomly, obliviously applies $\tau \uparrow A, B$
$(\mathsf{turn}, A)$ // reveals $\tau(\mathbb{A})$
let $\pi_{\tau(\mathbb{A})} \in \Pi$ be the permutation that sorts $\tau(\mathbb{A})$, i.e. $\pi_{\tau(\mathbb{A})}(\tau(\mathbb{A}))$ is the
  lexicographical minimum of $\{\pi(\tau(\mathbb{A})) \mid \pi \in \Pi\}$ w.r.t. a given order on the deck
  symbols
$(\mathsf{perm}, (\pi_{\tau(\mathbb{A})} \uparrow A) \circ (\pi_{\tau(\mathbb{A})} \uparrow B))$
```

Let $\pi_{\mathbb{A}} \in \Pi$ be the permutation that *sorts* $\mathbb{A}$, i.e., $\pi_{\mathbb{A}}(\mathbb{A})$ is the lexicographical minimum of $\{\pi(\mathbb{A}) \mid \pi \in \Pi\}$ w.r.t. a given order on the deck symbols[3]. The overall effect of $\mathsf{sort}_\Pi A{\uparrow}B$ should be that $\pi_{\mathbb{A}}$ is applied to both $\mathbb{A}$ and $\mathbb{B}$, yielding a sequence Γ' with $\Gamma'[A] = \pi_{\mathbb{A}}(\mathbb{A})$, $\Gamma'[B] = \pi_{\mathbb{A}}(\mathbb{B})$ and Γ' equal to Γ everywhere else. We permit B, and correspondingly $\mathbb{B}$, to be a sequence of k-element sequences $B = ((b^1_1 \dots, b^k_1), \dots, (b^1_n, \dots, b^k_n))$ for $k \in \mathbb{N}$, in which case applying $\pi_{\mathbb{A}}$ to $\mathbb{B}$ means applying $\pi_{\mathbb{A}}$ to each of the k sequences $\mathbb{B}_1 = \Gamma[(b^1_1, \dots, b^1_n)]$, ..., $\mathbb{B}_k = \Gamma[(b^k_1, \dots, b^k_n)]$.

Implementation of Sort Protocols

An example for a practical implementation is given in Fig. 2 and a formal specification in Protocol 1. The first step applies a randomly chosen permutation $\tau \in \Pi$ to A

[3] We use the order from $\mathbb{N}$ on cards with natural numbers, and $\clubsuit < \heartsuit$.

and B. Then, the cards in positions A are turned over, revealing $\tau(\mathbb{A})$ where $\mathbb{A}$ is the sequence of cards that was previously in positions A.

This allows us to recognize which permutation $\pi_{\tau(\mathbb{A})}$ would sort $\tau(\mathbb{A})$ and apply it to the sequences in positions A and B. Clearly, the overall effect is that $\mathbb{A}$ and $\mathbb{B}$ have both been permuted by the same permutation $\pi_{\tau(\mathbb{A})} \circ \tau$. Moreover, this permutation sorted the cards in positions A as desired.

If we only want to reset the sequence in A to a sorted one, i.e., without applying it to cards at positions B, (as in Protocols 11 and 12) we write $\mathsf{sort}_{\Pi} A$.

Definition 3.1 Let i be a index/step number of an action (or action sequence denoted by a shorthand, if you wish) in an execution of a protocol, and A a sequence of card positions of the protocol. Let $\mathsf{supp}\,(A, i) := \{\Gamma[A] \,:\, \Gamma$ is possible when reaching step $i\}$ be the set of possibilities for $\mathbb{A}$ when the protocol reaches the action at step i (before executing this step). We say an sub-protocol/shorthand $\mathsf{sort}_{\Pi} A {\uparrow} B$ at a step i is *valid* in a protocol if $\mathsf{supp}\,(A, i)$ is contained in an orbit O of the group action of Π on sequences, and $|O| = |\Pi|$.

The rationale behind this definition is that if $\mathsf{supp}\,(A, i)$ is subset of O w.r.t. Π, then shuffling $\mathbb{A}$ with Π destroys all information that is held in the sequence $\mathbb{A}$ prior to turning it. Thus, no information is leaked. The condition $|O| = |\Pi|$ ensures that the permutation $\pi_{\mathbb{A}} \in \Pi$ that sorts $\mathbb{A}$ is uniquely defined.[4]

Note that this slightly involved criterion is necessary to ensure security in the case that the permutation is chosen at random from a proper subset of S_n (on all n cards of the deck). An important example for this is a random cut, which we later use to apply a rotation encoded in a sequence. Assume for instance $\Pi = \langle(1\ 2\ 3)\rangle$ and $\pi \in \Pi$ uniformly random. Moreover, let X be the six-element set of permutations of (♡,♣,♠), and $s \in X$ be arbitrary. Revealing $\pi(s)$ to be, say, $\pi(s) =$ (♣,♡,♠) reveals, e.g., that s is not (♡,♣,♠). The reason is that Π has two orbits when acting on sequences of length 3 with symbols ♡,♣,♠ and we learn in which orbit we have been, excluding all sequences of the other orbit. This criterion is also suitable for achieving security, as shown by the following lemma.

Lemma 3.1 *If an shorthand/sub-protocol* $\mathsf{sort}_{\Pi} A {\uparrow} B$ *at step i is valid in a protocol, then the sequence revealed in the sub-protocol's turn step is independent of the random variable* Γ *denoting the card sequence before step i, and the random variable* Γ' *denoting the sequence directly after the sub-protocol.*

Proof By definition, $\mathsf{supp}\,(A, i)$ for the sub-protocol $\mathsf{sort}_{\Pi} A {\uparrow} B$ at step i is subset of an orbit O. Whatever the distribution of $\mathbb{A}$ is, if $\pi \in \Pi$ is chosen uniformly at random, then the sequence $\mathbb{A}' = \pi(\mathbb{A})$ revealed in the turn step is uniformly distributed on O.

[4] We could drop this condition without affecting security. The effect of sort would be that among all permutations that sort $\mathbb{A}$, one is chosen uniformly at random and applied to the cards in A and B.

It is thus independent of Γ. Since Γ' is a function of Γ, we conclude that $\mathbb{A}'$ is independent of (Γ, Γ'). □

Corollary 3.1 *If a protocol $\mathcal{P}$ contains no turn operations outside of valid instances of sort sub-protocols, then $\mathcal{P}$ is secure.*

Encoding Permutations

A sequence $(s_1, \dots, s_n) \in \{1, \dots, n\}^n$ of card symbols *encodes a permutation* π if $s_i = \pi(i)$ for $1 \le i \le n$. Let us denote $\mathcal{D}_5 := [1, 2, 3, 4, 5]$ and $\mathcal{D}_2 := [\clubsuit, \heartsuit]$, and give a short example.

Example 3.1 The 5-cycle permutation $\pi = (1\ 2\ 3\ 4\ 5)$ is represented via $\mathcal{D}_5$ by $\Gamma_\pi = (2, 3, 4, 5, 1)$. The (self-inverse) transposition $\tau = (1\ 2)$ is represented via $\mathcal{D}_2$ as $\Gamma_\tau = (\heartsuit, \clubsuit)$.

Useful Specializations. Two subclasses of sort protocols will be particularly useful. The first will be useful, e.g., to apply an encoded permutation to another sequence of cards, the second to rotate a sequence by a specified offset.

- Apply a permutation encoded in $\boldsymbol{A}$ to the sequence in $\boldsymbol{B}$. Assume that in a protocol, $\mathbb{A} = \Gamma[A]$ is known to always be a permutation of a fixed set M of n distinct cards, say of $M = \{1, 2, \dots, n\}$. Then, $\mathsf{sort}_{S_n} A \!\uparrow\! B$ is valid at this point i as $\mathsf{supp}\,(A, i)$ is a subset of all permutations of M, which is an orbit w.r.t. $\Pi = S_n$. The effect is that the permutation *encoded* in A is applied to $\Gamma[B]$. Whenever $\Pi = S_n$, we omit Π as an index of $\mathsf{sort}_\Pi A \!\uparrow\! B$.
- Apply a rotation encoded in $\boldsymbol{A}$ to the sequence in $\boldsymbol{B}$. Assume that in a protocol $\mathbb{A} = \Gamma[A]$ is known to always be a permutation of a multiset M with $n - 1$ copies of one symbol and one copy of another symbol, say $M = [(n-1)\cdot\heartsuit, \clubsuit]$. Let $\clubsuit < \heartsuit$ by convention. Then, for $\Pi = \langle (1\,2\,\dots\,n) \rangle$, an sort sub-protocol $\mathsf{sort}_\Pi A \!\uparrow\! B$ is clearly valid at this point i, as $\mathsf{supp}\,(A, i) \subseteq \{(\clubsuit,\heartsuit,\cdots,\heartsuit),(\heartsuit,\clubsuit,\heartsuit,\cdots,\heartsuit), \cdots, (\heartsuit,\cdots,\heartsuit,\clubsuit)\}$ and the latter is an orbit w.r.t. Π. The effect is that the rotation *encoded* in A is applied to $\Gamma[B]$. In this case, we also write $\mathsf{rot}\, A \!\uparrow\! B$ for $\mathsf{sort}_\Pi A \!\uparrow\! B$. (Note that this is similar to a part of the coupled rotation protocols given in [30].)

Note that for $n = 2$, the two cases are the same.

Non-destructive Variant sort^*. We define a variation sort^* of sort that differs only in so far as it should make no net change to the cards in positions A. For this, a sequence of *helping cards* is assumed to be available in (otherwise unused) positions $H = (h_1, \dots, h_n)$. We implement sort* in Protocol 2 by two applications of sort, where the latter restores $\mathbb{A}$ from the helping "register".

We say an application of sort* is *valid* whenever an application of sort would be valid and $\mathbb{H} := \Gamma[H] = (1, \dots, n)$ is guaranteed, i.e., H contains cards with numbers in ascending order. Note that sort* is defined as a shorthand or syntactic sugar via Protocol 2 in the same way as sort.

It is easy to see that under these conditions, if π is applied to the cards in positions A and H in the first sorting step, then π^{-1} is applied to the cards in positions A and H in the second sorting step, as this is the unique permutation that sorts the cards in positions H. Thus, one complete valid application of sort* makes no net changes to A and H. It is also easy to check that both applications of sort are valid in the original sense, therefore, Lemma 3.1 and Corollary 3.1 extend naturally to sort*. We use rot* for the variant using cyclic rotations.

Protocol 2. $\mathsf{sort}^*_\Pi(a_1, \dots, a_n) \uparrow (b_1, \dots, b_n)$:

```
Let $h_1, \dots, h_n$ be helping card positions (disjoint from $a_1, \dots, a_n$, $b_1, \dots, b_n$)
$\mathsf{sort}_\Pi\ (a_1, \dots, a_n) \uparrow ((b_1, h_1), \dots, (b_n, h_n))$
$\mathsf{sort}_\Pi\ (h_1, \dots, h_n) \uparrow (a_1, \dots, a_n)$
```

Stating Classical Protocols in Terms of sort

The standard AND, OR, XOR and COPY protocols due to Mizuki and Sone [37] can all be stated as single application of our sort sub-protocol as shown in Protocols 3 to 6 in Fig. 3. We also provide a *permutation application protocol* that takes the encoding of a permutation and a sequence as input and outputs the permuted sequence. This is in essence the permutation division protocol by Hashimoto et al. [22] (the only change being that we encode the inverse permutation). It has been suggested to us that more complex protocols, such as zero-knowledge protocols for Sudoku [44] and Makaro [6], as well as for the Millionaire's problem [34] can be interpreted to implicitly utilize our sort protocol. Moreover, the eight-card AND protocol for standard decks (where all card symbols are distinct) from [35] and the eight-card 3-bit majority protocol of [40] can be implemented using two sorts, the latter is given in Protocol 8.

Protocol 3. AND:

```
Input: bits $(x, y, 0)$ encoded in $((1,2),(3,4),(5,6))$
Output: encoding of $x \wedge y$
sort $(1,2) \uparrow ((3,4),(5,6))$ // swaps cards at $(3,4)$ (encoding $y$) with those at
    $(5,6)$ (encoding $0$), iff cards at $(1,2)$ are in reverse order (not sorted).
(result, $5, 6$)
```

Protocol 4. OR:

```
Input: bits $(x, y, 1)$ encoded in $((1,2),(3,4),(5,6))$
Output: encoding of $x \vee y$
sort $(1,2) \uparrow ((3,4),(5,6))$ // swaps cards at $(3,4)$ (encoding $y$) with those at
    $(5,6)$ (encoding $1$), iff cards at $(1,2)$ are in reverse order (not sorted).
(result, $3, 4$)
```

Protocol 5. XOR:

```
Input: bits $(x, y)$ encoded in $((1,2),(3,4))$
Output: encoding of $x \oplus y$
sort $(1,2) \uparrow (3,4)$ // swaps cards at $3$ and $4$ iff cards at $(1,2)$ are reversed.
(result, $3, 4$)
```

Protocol 6. n-COPY:

```
Input: bits $(x, 0, \ldots, 0)$ encoded in $((1,2), \ldots, (2n+1, 2n+2))$
Output: $n$ card pairs, all encoding $x$
sort $(1,2) \uparrow ((3,5,\ldots,2n+1),(4,6,\ldots,2n+2))$ // swaps cards at $3$ and $4$, and $5$
    and $6$, etc., until $2n+1$ and $2n+2$ iff cards at $(1,2)$ are reversed.
(result, $3, 4, \ldots, 2n+1, 2n+2$)
```

Protocol 7. APPLY:

```
Input: permutation $\pi$ encoded in $(1, \ldots, n)$
        and some sequence $\mathbb{A}$ in $(n+1, \ldots, 2n)$
Output: $\pi(\mathbb{A})$
sort $(1, \ldots, n) \uparrow (n+1, \ldots, 2n)$ // sorts cards at $1, \ldots, n$, and simultaneosly
    does the same card swappings in parallel to the (same-length) card
    sequence at $n+1, \ldots, 2n$.
(result, $n+1, \ldots, 2n$)
```

Fig. 3 The classical protocols AND, OR, XOR and COPY as well as a permutation application protocol, all stated as sort protocols

Protocol 8. 3-MAJORITY [NMS13]:

```
Input: bits $(x, y, 0, z)$ encoded in $((1,2),(3,4),(5,6),(7,8))$
Output: $\mathrm{maj}(x, y, z) := \begin{cases} 1, & \text{if } x + y + z \geq 2, \\ 0, & \text{otherwise.} \end{cases}$
sort $(1,2) \uparrow ((3,5),(4,6))$ // swaps cards at $3$ and $4$, and $5$ and $6$ iff cards at
    $(1,2)$ are reversed, i.e. XORs $x$ to registers of $y$ and $0$
sort $(3,4) \uparrow ((5,6),(7,8))$ // swaps cards at $(5,6)$ (encoding $x$) with those at
    $(7,8)$ (encoding $z$), iff cards at $(3,4)$ (encoding of $x \oplus y$) are reversed.
(result, $5, 6$)
```

Securely Evaluating a Universal Circuit

Let us start with the most direct case, namely implementing PFE using universal circuits, first constructed by Valiant [45]. We do not want to go into the details of the construction and just import facts about the general structure of the circuit and how it is used. In our examples, Alice provides her private function, here as a circuit C, and Bob his private input to the function, and it should hold that neither party learns anything about the other's respective secrets. The universal circuit U_n for circuits of size n takes as input an encoding $\langle C \rangle$ of C, where C has size n, and an input $I \in \{0,1\}^l$ of length l. We assume C to have fan-out and fan-in at most 2, i.e., each gate has at most two inputs and at most two outputs.

In the constructions by Valiant, U_n is described via a directed acyclic graph with $O(n \log n)$ vertices, where each vertex represents a logic gate taking values on its incoming edges as well as certain "configuration" (or programming) bits as input and computes outputs emitted to its outgoing edges. More concretely, U_n contains the following types of nodes:

- n *universal gates* with in- and out-degree exactly two and four configuration bits $c_1, \dots, c_4$ that compute

$$\mathsf{ug}(c_1, c_2, c_3, c_4, x, y) = (z, z), \text{ where } z = c_1\bar{x}\bar{y} + c_2\bar{x}y + c_3x\bar{y} + c_4xy$$

  where $c_1, \dots, c_4$ determine the Boolean operation performed at this gate, e.g., AND corresponds to $(c_1, \dots, c_4) = (0, 0, 0, 1)$.
- $O(n \log n)$ *X-switches* with a configuration bit c and in- and out-degree two, that compute

$$\mathsf{x}(c, a_0, a_1) = (a_c, a_{1-c}),$$

  where a_c is forwarded on one outgoing edge and a_{1-c} on the other.

- $O(n)$ *Y-switches* computing

$$\mathsf{y}(c, a_0, a_1) = a_c,$$

  where Alice's configuration bit c decides which of the two inputs is forwarded as the output.
- $O(n)$ *forks* (or "λ-switches") where the signal on one wire is forwarded to both outgoing wires, i.e., $\lambda(a) = (a, a)$.
- l input nodes with out-degree 1 and in-degree 0, and one output node with in-degree 1 and out-degree 0 with their natural interpretation.

The universal gates correspond to the gates of Alice's circuit with the configuration bits determining what kind of gate it is, and the configuration of X and Y-switches ensures that the intermediate results are routed correctly to the relevant gates. For us, it suffices that there is an (efficient) way to obtain $\langle C\rangle$ from C, which Alice applies beforehand. Valiant [45] describes such a general mapping from circuits C to a string of $O(n \log n)$ configuration bits for U_n, such that U_n configured with $\langle C\rangle$ (in canonical order) implements C.

We describe in Protocol 9 and Theorem 4.1 how, given U_n, encodings of $\langle C\rangle$ and Bob's input I in sequences of cards, we can compute $C(I)$ securely.

Theorem 4.1 *For any $l, n \geq 1$, there exists a secure card-based protocol $\mathcal{P}$ with the following properties*:

(i) *The input sequences are all sequences (V, P) where*

- *V encodes the values of l Boolean variables $(v_1, \dots, v_l) \in \{0,1\}^l$ using the deck $l\cdot[\clubsuit,\heartsuit]$.*
- *P encodes a circuit C of size n, via $k = O(n \log n)$ programming bits, i.e., via deck $k\cdot[\clubsuit,\heartsuit]$.*

(ii) *The output is two cards encoding $C(v_1, \dots, v_l)$.*

(iii) *In addition to the input cards, we use the helping deck $(m+1)\cdot[\clubsuit,\heartsuit]$, where $m = O(n)$ is the number of forks in U_n. (The additional pair is used for the* sort* *command.)*

(iv) *The protocol uses $x_n + y_n + 2f_n + 3u_n$ shuffles, where x_n is the number of X-switches, y_n is the number of Y-switches, f_n is the number of forks and u_n is the number of universal gates in U_n.*

Proof $\mathcal{P}$ is given as Protocol 9. All nodes of U_n are considered in some topological order $s_1, \dots, s_N$, allowing us to compute the bits "flowing" along each edge of U_n in a systematic way. The message at an edge e is stored in positions $V_e = (V_e[0], V_e[1])$. Note that the bit on each edge is only used in one subsequent computation: After processing s_i, only the bits on the edges crossing the cut $(\{s_1, \dots, s_i\}, \{s_{i+1}, \dots, s_N\})$ are needed in future computations. When processing s_{i+1} we may, therefore, when storing the bits for the outgoing edges of s_{i+1}, reuse the now freed up cards that stored the bits on the incoming edges of s_{i+1}. In Protocol 9, this is reflected by identifying V_e and $V_{e'}$ for some pairs (e, e') of edges. We only need a new pair of cards in the case of a fork.

To verify correctness, let us interpret the main sort commands in the protocol.

1. In the X-switch case, sort $C_v{\uparrow}(V_e, V_f)$ swaps the positions encoding the incoming input values at edges e and f, if the configuration bit of the X-switch equals 1 and leaves them unchanged, if it equals 0. This is exactly what we wanted.
2. In the Y-switch case, the command is exactly the same, with the difference that afterwards only the output bit that ends up in the first position (V_e) is used afterwards.

3. In the fork case, we (non-destructively, i.e., with restoring) copy the bit to another position, used as an additional output wire value.
4. The universal gate case is the most interesting. Recall that we want to evaluate $\mathsf{ug}(c_1, c_2, c_3, c_4, x, y) = (z, z)$ with $z = c_1\bar{x}\bar{y} + c_2\bar{x}y + c_3x\bar{y} + c_4xy$. For this, first observe that exactly one of the terms $\bar{x}\bar{y}$, $\bar{x}y$, $x\bar{y}$, xy equals one. Essentially, the values of x and y select which configuration bit constitutes the output. If $x = 0$ then only c_1 and c_2 are relevant. If $x = 1$ only c_3 and c_4 are. Therefore, in the first sorting step, we obliviously swap (C_1, C_2) for (C_3, C_4) if $x = 1$ and leave things as is, if $x = 0$. The interesting two configuration bits end up in positions C_1, C_2, without us knowing which they are.

   Now, we do the same with C_1, C_2, based on the value of y, so that the only relevant configuration bit is now in C_1. In the last step, we write this value in both V_g and V_h (recall the fan-out two requirement).

To see that $\mathcal{P}$ is secure, we use Corollary 3.1 and the fact that no turn operations are performed outside of sorting steps. □

Dependent on the topological ordering used in Protocol 9, the helping deck we use to implement forks is not fully required. Instead of using a "fresh" pair of cards to store a copy of the incoming value whenever a fork is encountered, we can reuse cards that have already served their function and will not be used in the remainder of the protocol. This includes, for instance, the cards that encoded configuration bits of universal circuits or X-switches that have already been executed.

Remark 4.1 (Reusability of the Circuit) If we would like to be able to execute the circuit multiple times, we want that the programming bits of Alice's program are not destroyed during the execution. Here, we have to take a little care to ensure that the relevant bits are written back and that conditionally swapped cards are "unswapped" again. For this variant of our algorithm, we replace all sort operations in Protocol 9 by their starred variants. In the case of v being a universal gate, we additionally need to take extra care: in the penultimate line of the case, instead of reusing V_e and V_f (which are now in temporary use to swap back the relative positions of the cards containing the configuration bits), we set V_g and V_h as the positions of two new cards, containing ♣♡ as in the fork case. To undo the swaps, we perform $\mathsf{sort}\ V_f{\uparrow}(C_1, C_2)$ and then $\mathsf{sort}\ V_e{\uparrow}((C_1, C_2), (C_3, C_4))$ at the very end of the procedure in the universal gate case. Afterwards, the cards in V_e and V_f may be reused again. Hence, this variant uses uses $2x_n + 2y_n + 2f_n + 8u_n$ shuffles, where x_n is the number of X-switches, y_n is the number of Y-switches, f_n is the number of forks and u_n is the number of universal gates in U_n.

Protocol 9. $\mathsf{UC}(\langle C\rangle, I)$: executing C on input I.
Note that the message at an edge e is stored in positions $V_e = (V_e[0], V_e[1])$.

```
foreach node $v$ of $U_n$ in (some) topological order do
    if $v$ is an input node then
        let $e$ be the outgoing edge and $I_e$ the positions of the corresponding input bit
        set $V_e := I_e$ // regard the cards at $I_e$ to belong to $e$
    else if $v$ is an X-switch then
        let $C_v$ be the position pair of the configuration bit for $v$
        let $e, f$ be the two incoming edges and $g, h$ the two outgoing edges
        $\mathsf{sort}\, C_v \uparrow (V_e, V_f)$
        set $V_g := V_e$ and $V_h := V_f$
    else if $v$ is a Y-switch then
        let $C_v$ be the position pair of the configuration bit for $v$
        let $e, f$ be the two incoming edges and $g$ the outgoing edge
        $\mathsf{sort}\, C_v \uparrow (V_e, V_f)$
        set $V_g := V_e$
    else if $v$ is a fork then
        let $e$ be the incoming edge and $g, h$ the two outgoing edges
        set $V_g := V_e$
        set $V_h$ as the positions of two new cards, containing ♣♡
        $\mathsf{sort}^*\, V_e \uparrow V_h$
    else if $v$ is a universal gate then
        let $C_1, \ldots, C_4$ be the position pairs containing the configuration bits of $v$
        let $e, f$ be the two incoming edges and $g, h$ the two outgoing edges of $v$
        $\mathsf{sort}\, V_e \uparrow ((C_1, C_2), (C_3, C_4))$
        $\mathsf{sort}\, V_f \uparrow (C_1, C_2)$
        set $V_g := V_e$ and $V_h := V_f$
        $\mathsf{sort}\, C_1 \uparrow ((V_g[0], V_h[0]), (V_g[1], V_h[1]))$
    else if $v$ is an output node then
        let $e$ be the only incoming wire at the output node
        $(\mathsf{result}, V_e)$
```

Securely Simulating a Turing Machine

Assume we wish to execute a Turing machine (TM) with a secret encoding provided by one player, Alice, on a secret input provided by another player, Bob. As any secure card protocol uses a fixed number of cards and has a runtime which is independent of the input, there must be known bounds on certain parameters of the Turing machine. Let *M* be a bound on the number of states, *N* a bound on the number of accessed tape cells and *t* a bound on the execution time. For simplicity, assume Alice's TM has precisely *M* states (it can be padded with dummy states),

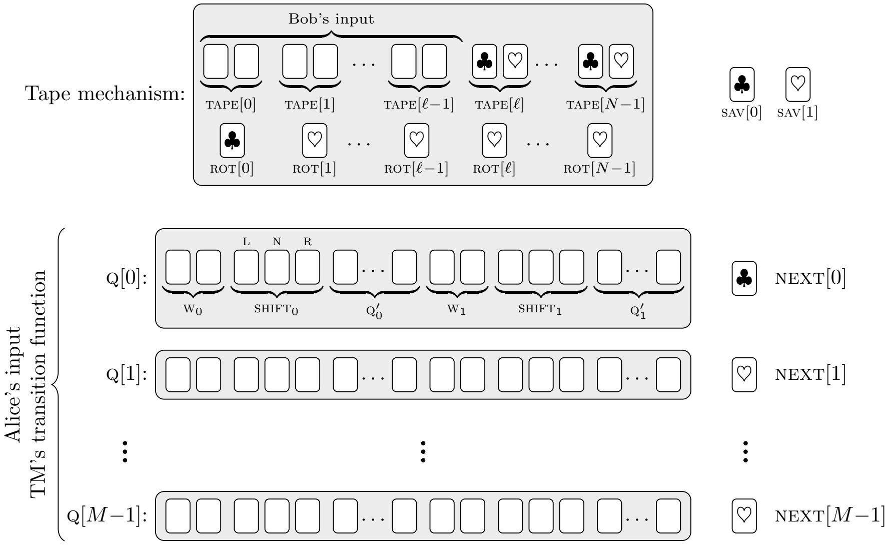


Fig. 4 Overview of a run of the universal TM

runs t steps ("halting" can be achieved by staying in one state, writing the current tape symbol and not moving) and think of the tape as a cycle of length N (which makes no difference for a TM only ever accessing N memory cells).

All cards (and names for them occurring in the following description) used for our protocol, with the exception of a few helping cards used for sort* and rot* operations, are given in Fig. 4. The encoding of a Turing machine consists of the encoding of its M states. The encoding of each state $q \in \{0, \dots, M-1\}$ consists of the encoding of two transitions, one for each of the two tape symbols ♡♣ and ♣♡. Take for instance the positions $\mathrm{W}_0, \mathrm{SHIFT}_0 = (\mathrm{L}, \mathrm{N}, \mathrm{R})$ and Q'_0 encoding the transition from state $q = 0$ if the tape symbol is ♣♡. The two cards in positions W_0 contain the tape symbol to be written. The three cards in positions SHIFT_0 specify the movement of the Turing machine head, ♣♡♡ for "left", ♡♡♣ for "right", ♡♣♡ for "no movement" / "halt". Lastly, the M cards in positions Q'_0 contain a unary encoding of $q - q' \pmod{M}$ where $q' \in \{0, \dots, M-1\}$ is the index of the state to be entered next (♣♡⋯♡ encodes 0, ♡♣♡⋯♡ encodes 1, etc.).

The input to the TM, provided by Bob, is encoded in the first l bits of the tape. When executing the Turing machine, the current tape cell will always be in position TAPE[0] and the current state in position Q[0]. Instead of having an explicit moving head we simply rotate the entire tape. Moreover, instead of having an explicit value encoding the current state, we rotate the sequence of states. This is also the reason we encode state index differences in the state transitions instead of absolute indices. The protocol is given as Protocol 10 and consists of a loop that does t times the following:

- "read" the tape symbol in position TAPE[0] by conditionally swapping the two transitions in state Q[0] such that the transition that should be done is available

in the positions W_0, SHIFT_0 and Q'_0. To undo this operation later, the value of TAPE[0] is also stored temporarily in (SAV[0], SAV[1]).
- the content of TAPE[0], which was reset to 0 in the previous step, is now overwritten with the symbol in position W_0.
- The cards in positions (L, N, R) are used to rotate the ♣ of ROT[0] into the positions ROT[0], ROT[1] or ROT[N−1] depending on whether the ♣-card among SHIFT_0 is in position N, R or L, respectively. Then, the TAPE and ROT cards are rotated together such that the tape cell whose corresponding ROT card is ♣ comes to rest in position TAPE[0] (and such that one does not learn which rotation has been performed.)
- The same idea is used to first copy the information about the next state into NEXT[0 ... M−1] and then rotate the sequence of all states accordingly. Note that we need to undo the conditional swap of the two transitions in $Q[0]$ before the rotation of the states (using a coupled sorting with (SAV[0], SAV[1])).

Protocol 10. executeTM()
The registers TAPE, W_i, SHIFT_i, Q'_i, SAV, ROT, NEXT, Q are given in Fig. 4.

```
repeat $t$ times
    sort TAPE[0] $\uparrow$ ((W$_0$, SHIFT$_0$, Q$'_0$, SAV[0]), (W$_1$, SHIFT$_1$, Q$'_1$, SAV[1]))
    sort* W$_0$ $\uparrow$ TAPE[0] // write symbol from W$_0$ into TAPE[0]
    rot* (N, L, R) $\uparrow$ (ROT[0], ROT[1], ROT[$N-1$])
    rot ROT $\uparrow$ TAPE // rotate tape according to ROT
    rot* Q$'_0$ $\uparrow$ NEXT // write next state (relative to current) into NEXT
    sort SAV $\uparrow$ ((W$_0$, SHIFT$_0$, Q$'_0$), (W$_1$, SHIFT$_1$, Q$'_1$))
    rot NEXT $\uparrow$ Q // update state by rotating the state table according to
        NEXT offset
result TAPE // or parts of it
```

Using this protocol idea, we obtain the following theorem.

Theorem 5.1 *For any $l \geq 0, N, M, t \geq 1$, there exists a secure card-based protocol $\mathcal{P}$ with the following properties*:

(i) *The input sequences are all sequences* (V, P) *where*

- *V encodes the values of l Boolean variables $(v_1, \dots, v_l) \in \{0,1\}^l$ using the deck* $l \cdot$ [♣,♡].
- *P encodes a Turing machine T with a state set of size M, using the deck* $2M \cdot$ [$3 \cdot$ ♣,$(M+2) \cdot$ ♡].

(ii) *The output is a sequence of cards encoding the output of T after running t steps on a cyclic tape of length N initially containing the input* $(v_1, \dots, v_l)$.

(iii) *In addition to the cards encoding the inputs, the helping deck* [$(N-l+3) \cdot$ ♣, $(M+2N-l-1) \cdot$ ♡] ∪ [♣, $\min\{2, M-1\} \cdot$ ♡] *is used. (The latter part is implicit in the use of the starred* rot* *commands and not shown in Fig.* 4.)

(iv) *The protocol uses* $10t$ *shuffles.*

Proof The protocol is given in Protocol 10 and Fig. 4. For security, observe that the protocol consists only of sort sub-protocols; we can thus use Corollary 3.1.

For the cards needed, we just count the number of cards depicted in Fig. 4. In a bit more detail, for the helping cards needed, note that we need $N - l$ pairs of ♣♡ for the empty tape cells, which are placed next to Bob's input string. We have one ♣ for each of the registers ROT, SAV and NEXT, and $N - 1$, 1 and $M - 1$ ♡s, respectively. The second part of the union scales with the size of the largest register to be used in starred commands, which is either SHIFT$_0$ or Q$'_0$. □

Remark 5.1 (Variants to the Implementation) Using techniques presented in Sect. 6, we could use a binary instead of a unary encoding of state indices in the encoding of transitions. This would reduce the number of required cards from $O(N + M^2)$ to $O(N + M \log(M))$. However, given that the charm of Turing machines is their simplicity rather than their efficiency, we felt that we should reserve this trick for later.

For simplicity, we also chose to describe how to implement TMs with band alphabet $\{0, 1\}$, excluding the special blank symbol ␣. While one can generically map this to the standard case by using an encoding $1 \hat{=} 11$, $0 \hat{=} 10$, and ␣$\hat{=} 00$, let us briefly discuss how one can easily upgrade our implementation with a TM supporting an additional blank symbol. For this, we encode tape cells with three cards via ♣♡♣ $\hat{=}$ 0, ♡♣♣ $\hat{=}$ 1 and ♣♣♡ $\hat{=}$␣. In this way, the first two cards encode the value as previously, unless they are ♣♣, which would be a blank. We then need to add W$_2$, SHIFT$_2$ and Q$'_2$ to each of the Qs, specifying the operation in the case that a blank symbol is used (Note that the W$_i$ contain the symbol to be written in reversed order, to ensure the right action is done to the tape cards). This approach has the advantage of allowing us to learn the length of the output after the computation (if it is not to be protected), by just turning over the third card in each of the tape cells and outputting (the first two cards of) those cells which do not show a ♡, i.e., which are not blank.

Remark 5.2 (Reusability of the TM) First note that we never destroy any of the state description entries of the TMs code as in normal execution it is always possible to enter the state again. Hence, to be able to run a TM multiple times, we only need to ensure that after the execution the first state is again in Q[0]. As we cannot trust Alice to provide a program that guarantees this behavior, we can introduce an additional register START[0...$M - 1$] which is a copy of NEXT and is rotated together with Q. It can then be used to rotate Q back into its initial configuration by executing rot START↑Q after the loop in Protocol 10. Hence, this variant uses $10t + 1$ shuffles. (Resetting all tape cells to 0 and placing the new input is excluded here, but can be easily appended.)

Securely Simulating a Random Access Machine (RAM)

We now describe a simple bounded Random Access Machine model. The goal is to execute a RAM machine with a secret encoding of the machine specified by one player, Alice, on a secret input provided by another player, Bob.

A Simple RAM Model

We assume fixed constants $N = 2^n$ (memory words), $M = 2^m$ (instruction groups), $l \leq N$ (input size) and $t < \infty$ (time limit). The machine has access to N binary words $\text{RAM}[0], \dots, \text{RAM}[N-1]$ of length n each, the first l of which contain the input and the remaining $N - l$ contain zero. The following types of instructions are available, where x, y are n-bit words and p is an m-bit word:

Load a Constant. $\text{RAM}[x] \leftarrow y$
Copy. $\text{RAM}[x] \leftarrow \text{RAM}[y]$
Indirect Read. $\text{RAM}[x] \leftarrow \text{RAM}[\text{RAM}[y]]$
Indirect Write. $\text{RAM}[\text{RAM}[x]] \leftarrow \text{RAM}[y]$
Addition. $\text{RAM}[x] \leftarrow \text{RAM}[x] + \text{RAM}[y]$
Subtraction. $\text{RAM}[x] \leftarrow \text{RAM}[x] - \text{RAM}[y]$
Conditional Jump. jnz $\text{RAM}[x]\ p$

To simplify the implementation step later, we assume that a program is a sequence $\text{I}[0], \dots, \text{I}[M]$ of *groups* of instructions. Each group of instructions contains precisely one instruction of each of the above types, in canonical order. Note that this fixed instruction order does not affect the strength of the model. Indeed, if we assume that without loss of generality the cell $\text{RAM}[0]$ is never used in any "real" instruction, we may choose $x = y = 0$ to turn any instruction into a dummy instruction that has no effect. By turning all but one desired instruction in each instruction group into such a dummy instruction, we can implement programs without having to worry about the fixed instruction order at the expense of increasing the number of instructions by a constant factor.

Here, the jnz $\text{RAM}[x]\ p$ ("jump if not zero") instruction means that if $\text{RAM}[x]$ contains zero, the execution should continue with the next instruction group. Otherwise, p is to be interpreted as the relative offset to the next instruction group that should be executed, i.e., if the current instruction group has index j, then the instruction group with index $(j + p) \bmod M$ should be executed next.

Implementation with Cards

Assume we want a secure implementation of the RAM model with parameters $N = 2^n$, $M = 2^m$, l, t using playing cards. We may imagine that one player, Alice, provides the sequence of instructions, and the other player, Bob, provides the input in $\text{RAM}[0 \dots l{-}1]$ of $l{\cdot}n$ bits. As usual, each bit is encoded with a pair of cards and a word of n or m bits is a sequence of n or m such pairs. In addition to the inputs, we have an encoding of $\text{RAM}[l \dots N{-}1]$ (initially zero) and two additional n-bit

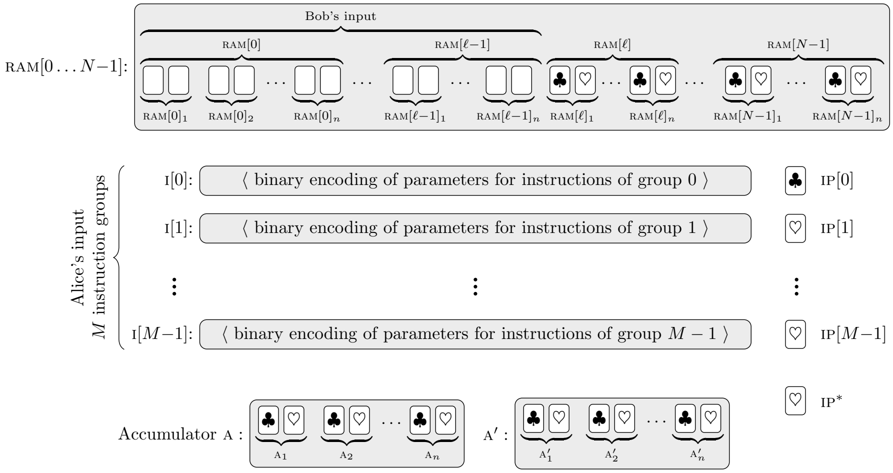


Fig. 5 Overview of our RAM machine construction, cf. Protocol 13

"accumulators" A and A' (initially zero). Finally, there are ♡-cards in ("instruction pointer") positions labeled IP[1], …, IP[$M-1$], IP* and one ♣-card in the position labeled IP[0], which will be used for the conditional jumps. An overview is given in Fig. 5.

We say a few words about the implementation of the instructions, starting with a general description of how words can be loaded from and stored to arbitrary addresses.

Loading a Word. Assume that an address is available as an n-bit word $x = (x_1, \dots, x_n)$, each bit x_i encoded as a pair of face-down cards in positions $X_i = (X_i[0], X_i[1])$ and that the word RAM[x] should be loaded into the accumulator. We give an implementation as Protocol 11. The first loop uses n conditional swaps of RAM ranges to transport the content of RAM[x] into RAM[0]. The invariant is that after the i-th loop, the content of RAM[x] has been transported to RAM[$x \,\&(2^{n-i} - 1)$] where & denotes the bitwise AND. For instance, if $n = 4$ and $x = 10 = (1010)_2$, then in the rounds $i = 1$ the left half RAM[0 … 7] and right half RAM[8 … 16] of the memory would be swapped and in round $i = 3$ the ranges RAM[0, 1] and RAM[2, 3] would be swapped, in total transporting RAM[10] via RAM[2] to RAM[0].

The second for-loop copies the content of RAM[0] to the accumulator. Since the copy protocol can copy information only onto card pairs that are in a known state, we must securely reset the accumulator bits before each copy operation. The third for-loop undoes all swaps of the first loop, in reverse order. In total, this uses $7n$ shuffles.

Protocol 11. load(X), where $X = (X_1, \ldots, X_n)$ is a sequence of n card-pairs encoding an n-bit address $x = (x_1, \ldots, x_n)$:

```
for $i = 1$ to $n$ do
    sort* $X_i \uparrow (\text{RAM}[0 \ldots 2^{n-i}-1], \text{RAM}[2^{n-i} \ldots 2^{n-i+1}-1])$
for $i = 1$ to $n$ do
    sort $A_i$ // securely reset $i$-th bit of accumulator
    sort* $\text{RAM}[0]_i \uparrow A_i$ // copy $i$-th bit
for $i = n$ down to $1$ do
    sort* $X_i \uparrow (\text{RAM}[0 \ldots 2^{n-i}-1], \text{RAM}[2^{n-i} \ldots 2^{n-i+1}-1])$
```

Storing a word. Storing is very similar to loading, we give an implementation in Protocol 12. Here, instead of copying the RAM content to the accumulator in the second line of the second for loop, we copy the value of the accumulator into the RAM. As above, this uses $7n$ shuffles.

Protocol 12. store(X), where $X = (X_1, \ldots, X_n)$ is a sequence of n card-pairs encoding an n-bit address $x = (x_1, \ldots, x_n)$:

```
for $i = 1$ to $n$ do
    sort* $X_i \uparrow (\text{RAM}[0 \ldots 2^{n-i}-1], \text{RAM}[2^{n-i} \ldots 2^{n-i+1}-1])$
for $i = 1$ to $n$ do
    sort $\text{RAM}[0]_i$ // securely destroy content
    sort* $A_i \uparrow \text{RAM}[0]_i$ // copy $i$-th bit of accumulator
for $i = n$ down to $1$ do
    sort* $X_i \uparrow (\text{RAM}[0 \ldots 2^{n-i}-1], \text{RAM}[2^{n-i} \ldots 2^{n-i+1}-1])$
```

Move operations. The operations previously dubbed **copy**, **indirect read** and **indirect write** are easy to implement using the load and store algorithms. For temporary storage, the accumulator A' is used. For instance, the indirect write operation RAM[RAM[x]]←RAM[y] with the words x and y encoded in positions X and Y can be implemented using load(Y), swap(A, A'), load(X), swap(A, A'), store(A'), where swap just swaps the two card sequences. As each load or store operation uses $7n$ shuffles, we use $14n$ shuffles for copy, and $21n$ shuffles for indirect read and indirect write.

Loading Constants. Copying a value given directly in the instruction is simply done by copying each of the n bits one by one. This uses $7n$ shuffles.

Addition and Subtraction. Secure half and full adders have been described by [33]. If $n \geq 2$, the accumulator A' is sufficient to store carry-bits temporarily. Note that both protocols use $5n$ shuffles (more precisely, random bisection cuts), as subtraction uses the full adder with the carry bit set to 1 and all bits of the second number inverted (via a simple perm operation). We omit the details.

Conditional Jump. While it would be possible to have an instruction pointer that is affected by jump operations, we opt for an approach that seems slightly more elegant. We always execute instruction group I[0], and when executing the last instruction jnz RAM[x] p of that group, we rotate the sequence of all instructions such that *either* IP[1] *or* IP[p] becomes IP[0], depending on the value of RAM[a]. See below for the exact description. Counting the shuffles in the relevant part of Protocol 13 yields $8n + 2m + 2$ shuffles, as the n bit OR operation uses n shuffles. (Note that here p might even be 0, meaning that if RAM[x] $\neq 0$ the same instruction group is repeated again. Due to the time limit t, this cannot result in a real infinite loop and hence would not exhibit unusual detectable behavior that, e.g., Alice could use to learn information on Bob's input.)

The overall execution of the RAM program is given in Protocol 13. We assume the addresses x and p are available in positions X and P, respectively. To carry out theconditional jump, first load x into the accumulator and form the Boolean OR of all its bits. Assuming RAM[0] is not zero, then the bit a_1 is set to true by this OR operation and the single ♡-card is swapped into IP* before the for-loop and is put into position IP[1] afterwards. If, however, RAM[0] is zero, then a_1 is set to false in which case the for-loop transports the ♣-card into position IP[p] (the loop invariant is that the ♣-card is in position IP[$p\,\&(2^{m-i} - 1)$]). The rot operation in the last step rotates the sequence of instructions as desired.

Protocol 13. executeRAM():

```
repeat $t$ times
    ⟨execute all instructions in group I[0], except the jump⟩
    // Now execute jnz RAM[$x$] $p$:
    load($X$)
    $A_1 \leftarrow A_1$ OR $A_2$ OR ... OR $A_n$
    $A_1 \leftarrow \neg A_1$ // swap $A_1$'s cards
    sort* $A_1 \uparrow$ (IP[0], IP*)
    for $i = m$ down to $1$ do
        sort* $P_i \uparrow$ (IP[0...$2^{m-i}-1$], IP[$2^{m-i}$...$2^{m-i+1}-1$])
    sort $A_1 \uparrow$ (IP*, IP[1])
    rot IP[0...$M-1$] $\uparrow$ I[0...$M-1$]
result RAM // or parts of it
```

Theorem 6.1 *For any $N = 2^n, M = 2^m, l < N, t \geq 1$, there exists a secure card-based protocol $\mathcal{P}$ with the following properties*:

(i) *The input sequences are all sequences* (V, P) *where*

- *V encodes l n-bit words $(v_1, \ldots, v_l) \in \{0, 1\}^{nl}$ using the deck $nl\cdot$[♣,♡].*
- *P encodes an n-bit-word RAM machine R with M instruction groups using the deck $kM\cdot$ [♣,♡], where $k = O(n + m)$ is the length of the encoding of one instruction group.*

(ii) *The output is a sequence of cards encoding the output of R on input* $(v_1, \dots, v_l)$ *after t steps.*

(iii) *In addition to the cards encoding the inputs, we need the helping deck* $(N - l + 2)n\cdot$ [♣,♡] ∪ [♣, $M\cdot$ ♡]. (*Additional cards for the starred sort variants can borrow from* A'.)

(iv) *The protocol uses* $(85n + 2m + 4)t$ *shuffles.*

Proof For the correctness, we refer to the above explanation of all the relevant commands. For security we again use Corollary 3.1 and the fact that we do not turn over any cards outside sort or rot operations. For this, note that the OR operation in line 5 of Protocol 13 can be framed as a sort operation, cf. Protocol 4. The number of shuffles is derived by counting the numbers of shuffles in each instruction type as specified above. This yields $(7n + 14n + 21n + 21n + 5n + 5n + 8n + 2m + 4)t = (85n + 2m + 4)t$ shuffles. □

Remark 6.1 (Reusability of the Program) Similarly to Remark 5.2 for the TM case, we can ensure that we end in the original configuration (with the first instruction in IP[0]) by introducing an additional register START[0...$M - 1$] which is rotated together with the instruction groups and IP. At the end of the execution, we use it to rotate everything back into place and additionally reset the accumulators. This variant uses an additional $2n + 1$ shuffles (again not including the reset of the RAM cells and providing the new input).

Securely Evaluating a Branching Program

Branching Programs [4] are commonly used for constructing program obfuscation, e.g., in [17, 20, 47], which inspired this section.

Branching Program. A *branching program B* of length N and width w for l variables is a sequence $((j^{(i)}, \pi_0^{(i)}, \pi_1^{(i)}))_{1\le i\le N} \in (\{1, \dots, l\} \times S_w \times S_w)^N$ of *instructions*. The permutation belonging to a sequence $\vec{v} = (v_1, \dots, v_l) \in \{0, 1\}^l$ of inputs is

$$B(\vec{v}) = \prod_{1\le i\le N} \pi^{(i)}_{v_{j(i)}}.$$

In other words, in the i-th step, the value of the $j^{(i)}$-th variable determines which of the two permutations of the i-th instruction is used.

For $\sigma \in S_w$, we say *B* σ*-computes a Boolean circuit C*, if for any $\vec{v} \in \{0, 1\}^l$

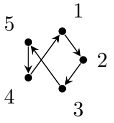
$$B(\vec{v}) = \begin{cases} \sigma, & \text{if } C(\vec{v}) = 1, \\ \mathsf{id}, & \text{if } C(\vec{v}) = 0. \end{cases}$$

Now let State be a set of states on which S_w acts via some group action $*$ and executing B on $\vec{v}$ starting from some start state $q_0 \in \mathsf{State}$ means computing states $(q_i)_{1\le i\le N}$ iteratively as $q_{i+1} = \pi_{v_{j(i)}} * q_i$. Of course, we end with $q_N = \pi_{v_{j(N)}} * \dots * \pi_{v_{j(1)}} * q_0 = B(\vec{v}) * q_0$.

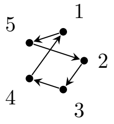
In this paper, State is a set of card sequences of length w and $\pi * q$ yields the card sequence q permuted by π.

A Peculiar Subset of S_5. Barrington's Theorem makes heavy use of the fact that S_5 is not a solvable group. In particular, there are permutations $\pi, \tau \in S_5$ such that the commutator $[\pi, \tau] := \pi \circ \tau \circ \pi^{-1} \circ \tau^{-1}$ is not the identity permutation. There is some freedom when choosing permutations for the construction that follows. To be more specific, we define the five permutations $\varphi_0, \dots, \varphi_4$ as

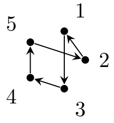
$\varphi_0:$



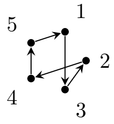
$\varphi_1:$



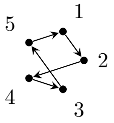
$\varphi_2:$



$\varphi_3:$



$\varphi_4:$



In general, we can define $\varphi_i = (1\ 2\ 3\ 4\ 5)^i \circ \varphi_0 \circ (1\ 2\ 3\ 4\ 5)^{-i}$ for any $i \in \mathbb{Z}$ but, of course, only the remainder of the index modulo 5 is relevant.

It is easy to check that $\varphi_0 = \varphi_5 = [\varphi_3, \varphi_4]$ and $\varphi_0^{-1} = \varphi_5^{-1} = [\varphi_1, \varphi_3]$. We can, therefore, write each element $\varphi \in F := \{\varphi_0, \dots, \varphi_4, \varphi_0^{-1}, \dots, \varphi_4^{-1}\}$ as $\varphi = [\varphi', \varphi'']$ for some other elements $\varphi', \varphi'' \in F$. More concretely, we have

$$\varphi_i = [\varphi_{i+3}, \varphi_{i+4}], \qquad \varphi_i^{-1} = [\varphi_{i+1}, \varphi_{i+3}].$$

Barrington's Theorem. We now state a central theorem due to Barrington, which we specialize to permutations from the set F defined above. For self-containedness and illustration, we give the elegant and constructive proof in full. Recall from Sect. 2 that the depth of a circuit C is the maximum number of $\wedge$ and $\vee$ gates on a path in C.

Theorem 7.1 (Barrington [4]) *For any Boolean circuit C of depth d and $\varphi \in F$ there exists a branching program $B = B(C)$ of width 5 and $N \leq 4^d$ instructions that φ-computes C.*

Proof The proof works by induction on the length d' of the longest path in C. If $d' = 0$, then we also have $d = 0$ and the output node is labeled with a constant 0, a constant 1 or the index j of a variable. In these cases, the trivial branching programs with a single instruction of the form $(\_, \mathsf{id}, \mathsf{id})$, $(\_, \varphi, \varphi)$ or $(j, \mathsf{id}, \varphi)$, respectively, φ-compute C (here, $\_$ is a placeholder for an arbitrary variable index).

Now assume $d' > 0$. If the output node is labeled „$\neg$", then the value at its unique predecessor is computed by a circuit C' with longest path of length $d' - 1$. Therefore, there is a branching program B' that φ^{-1}-computes C' with at most 4^d instructions. Let (j, π, π') be the last instruction of B'. Replacing it with $(j, \varphi \circ \pi, \varphi \circ \pi')$ yields a branching program B that φ-computes C since we have

$$B(\vec{v}) = \varphi \Leftrightarrow B'(\vec{v}) = \mathsf{id} \Leftrightarrow C'(\vec{v}) = 0 \Leftrightarrow C(\vec{v}) = 1$$

and for similar reasons $B(\vec{v}) = \mathsf{id} \Leftrightarrow C(\vec{v}) = 0$.

If the output node is labeled $\wedge$, then values at its two predecessors are computed by two circuits C' and C'' with longest path of length at most $d' - 1$ and depth at most $d - 1$. We previously observed that we can write $\varphi = [\varphi', \varphi'']$ for two permutations $\varphi', \varphi'' \in F$. Let $B'_{\varphi'}$ and $B'_{\varphi'^{-1}}$ be two branching programs that φ'-compute and φ'^{-1}-compute C', respectively, and similarly $B''_{\varphi''}$ and $B''_{\varphi''^{-1}}$ be two branching programs that φ'-compute and φ''^{-1}-compute C'', respectively.

We obtain B as the concatenation of these four branching programs. Depending on the values $r' = C'(v_1, \dots, v_l)$ and $r'' = C''(v_1, \dots, v_l)$ we get the following behavior of B:

$$\begin{aligned} B(\vec{v}) &= B'_{\varphi'}(\vec{v}) \circ B''_{\varphi''}(\vec{v}) \circ B'_{\varphi'^{-1}}(\vec{v}) \circ B''_{\varphi''^{-1}}(\vec{v}) \\ &= \begin{cases} \varphi' \circ \varphi'' \circ \varphi'^{-1} \circ \varphi''^{-1} = [\varphi', \varphi''] & = \varphi \text{ if } r' = r'' = 1 \\ \mathsf{id} \circ \varphi'' \circ \mathsf{id} \circ \varphi''^{-1} & = \mathsf{id} \text{ if } r' = 0, r'' = 1 \\ \varphi' \circ \mathsf{id} \circ \varphi'^{-1} \circ \mathsf{id} & = \mathsf{id} \text{ if } r' = 1, r'' = 0 \\ \mathsf{id} \circ \mathsf{id} \circ \mathsf{id} \circ \mathsf{id} & = \mathsf{id} \text{ if } r' = r'' = 0 \end{cases} \end{aligned}$$

Since $C(\vec{v}) = 1 \Leftrightarrow r' = r'' = 1$, this means B indeed φ-computes C. □

Implementing Branching Programs with Cards

We first describe how the encoding $P = P(C)$ is obtained from C, as the format of P already contributes to hiding details about C, especially the pattern in which variables are used. Firstly, by Barrington's Theorem (Theorem 7.1) there is a branching program $B = B(C)$ that φ_0^{-1}-computes C with $N \leq 4^d$ instructions. We now transform B into a *normalized branching program* B' by preceding each instruction (j, π_0, π_1) of B with the $j - 1$ dummy instructions $(1, \mathsf{id}, \mathsf{id}), \dots, (j - 1, \mathsf{id}, \mathsf{id})$ and appending to it the $l - j$ dummy instructions $(j + 1, \mathsf{id}, \mathsf{id}), \dots, (l, \mathsf{id}, \mathsf{id})$. This means that B' accesses all variables periodically in canonical order. Note that B' contains $lN \leq l{\cdot}4^d$. (In addition, we may choose to pad B' to a longer program B'' of length lN' if we wish to hide the length of B' and thus of B.) Clearly, B' exhibits the same behavior as B. The sequence P is now simply obtained by concatenating the lN sequences encoding the permutations occurring in the description of B'.

Theorem 7.2 *For any $l, N \geq 1$, there exists a secure card-based protocol $\mathcal{P}$ with the following properties*:

(i) *The input sequences are all sequences (V, P) where*

– *V encodes the values of l Boolean variables $(v_1, \dots, v_l) \in \{0, 1\}^l$ using the deck $l{\cdot}$ [♣,♡].*

- *P encodes a normalized branching program B of length lN with one bit output using the deck* $2lN{\cdot}[1,2,3,4,5]$.

(ii) *The output is two cards encoding $B(v_1,\dots,v_l)$.*

(iii) *In addition to the cards encoding the inputs, the helping deck* [2·♡,5·♣] *is used.*

(iv) *Each execution of the protocol performs* $3lN$ *shuffle actions.*

Proof The protocol is described in Protocol 14. We denote by capital letters the sets of positions on which the corresponding parts of the input (denoted by lower case letters) are present at the start of the protocol. Additionally, there are helping cards present in positions Q that initially contain the sequences ♣♡♣♣♣ as well as two cards to support the sort*-operation (not shown in Fig. 6).

Protocol 14. Executing a branching program.

```
for $i \leftarrow 0$ to $N-1$ do
    for $j \leftarrow 1$ to $\ell$ do
        sort* $V_j \uparrow (\Pi_0^{(i\ell+j)}, \Pi_1^{(i\ell+j)})$
        sort $\Pi_0^{(i\ell+j)} \uparrow \mathsf{Q}$
result $\mathsf{Q}_R$
```

Consider an iteration of the inner loop with $k = li + j$. First, the encodings of the two permutations $\pi_0^{(k)}$ and $\pi_1^{(k)}$ (in positions $\Pi_0^{(k)}$ and $\Pi_1^{(k)}$) are swapped if v_j (in position V_j) is 1 and left as is otherwise. Hence, an encoding of $\pi_{v_j}^{(k)}$ ends up in position $\Pi_0^{(k)}$, from where it is obliviously applied to the sequence in Q. For correctness, note that by assumption the normalized branching program φ_0^{-1}-computes C, i.e., if the output is 0, in total we perform id on the cards in Q, which results in a 0 being encoded in Q_R. If C outputs 1, then φ_0^{-1} is applied to the cards of Q, resulting in ♡♣♣♣♣, as φ_0^{-1} maps $2 \mapsto 1$, yielding an encoded 1 in Q_R.

Security of $\mathcal{P}$ follows again from the fact that the protocol is only composed by valid sort operations and Corollary 3.1. □

Remark 7.1 (Reusability of the Program) To allow for reusing the branching program after its execution, we would need to write the executed permutation of each step back into its register and to undo any conditional swaps. In more formal terms, we replace the sort command in the second line of the inner loop of Protocol 14 with its starred variant. To undo the swap, we repeat the first line of the inner loop after the second line. Moreover, we reset the register Q. Hence, this variant of the protocol uses $6lN + 1$ shuffles.

A Note Regarding Active Security. Note that a malicious Alice might learn something about the input passed to the program by choosing the permutations of the

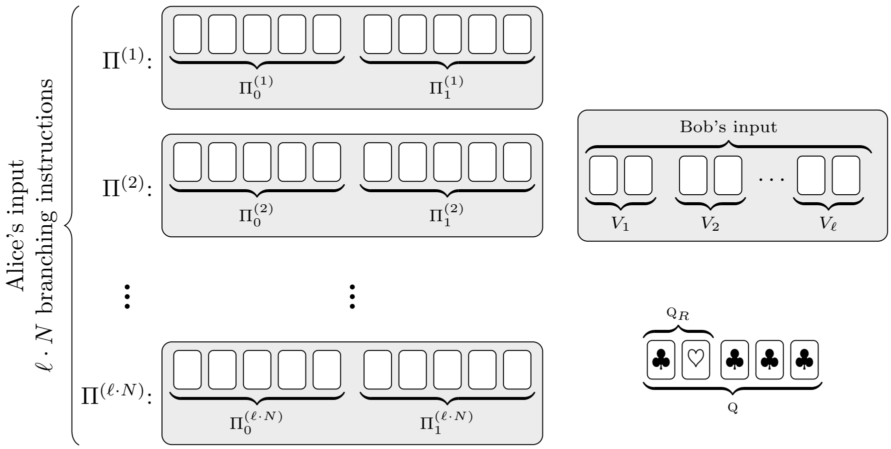


Fig. 6 Overview of the branching program construction. Alice's input is the branching program $((j^{(i)}, \pi_0^{(i)}, \pi_1^{(i)}))_{1 \le i \le N} \in (\{1, \dots, l\} \times S_5 \times S_5)^N$ in normalized form

program in such a way that the output (the first two cards in Q after the protocol run) is not ♣♡ or ♡♣, but ♣♣. If we want to avoid this, we can initialize q_0 with ♣♡♣♡♣ (replacing the penultimate ♣ with a ♡), and instead of opening just the first two cards at the end, we have to ensure that the content of the register gets mapped to a single bit, without revealing anything else. For this, note that after a protocol run of a legal program, Q contains one of two configurations namely ♣♡♣♡♣ if id was applied, and ♡♣♣♣♡ if φ_0^{-1} was applied. Important here, is that in the first case, the ♡ s have distance 1 and in the second case distance 0, which is invariant over random cuts, and represents the two possible configuration classes (orbits w.r.t. random cuts) in the five-card trick [8]. We cannot use the five-card trick directly, as its output is not in committed format, however. To overcome this, we can make use of the five-card AND protocol of [3], which starts with a situation as above and then outputs a bit commitment to the AND value in a (restart-free) Las Vegas fashion. (Note that this protocol is shown to be optimal/card-minimal in a strong sense in [27].) This change would add seven shuffles (five random cuts and two random bisection cuts) in expectation.

Moreover, for active security in all the protocols in this paper, one should additionally implement the shuffle operation with active security as in [30]. For ease of implementing the coupled shuffles, we recommend to use envelopes to avoid additional helping cards, as in Fig. 2.

Conclusion

We give four card-efficient and conceptually simple protocols for executing a universal machine model in a secure multiparty computation protocol, hence achieving Private Function Evaluation. These are for circuits, Turing and word-RAM machines and branching programs, giving the user a palette of options, from which they can choose the most suitable one. As an interesting building block—also largely simplifying security proofs—we introduce sort protocols, which we believe to be of independent interest, as many protocols from the literature can be restated in these terms. We give the concrete numbers of necessary cards for each of the models, carefully reusing helping cards where possible. We additionally discuss several adaptations, e.g., on how to execute these in a non-destructive way that lets us reuse the program multiple times.

Our results can also be interpreted as a straightforward instantiation of Oblivious RAM (ORAM), making heavy use of the fact that we can physically and obliviously move around "RAM cells", which is not possible in the usual cryptographic ORAM model. By stating these classical cryptography problems, such as constructing ORAM or program obfuscation in the language of card-based cryptography, it might not only be of didactic use in explaining these to students, but also provide some insight into the constructions in the classical cryptographic realm.

Acknowledgements We would like to thank the anonymous reviewers. Alexander Koch was supported by the Competence Center for Applied Security Technology (KASTEL).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abbott, R.: Eleusis and Eleusis Express. http://www.logicmazes.com/games/eleusis/ (visited on 10/02/2018)
2. Achenbach, D., Borcherding, A., Löwe, B., Müller-Quade, J., Rill, J.: Towards Realising Oblivious Voting. In: Obaidat, M.S. (ed.) E-Business and Telecommunications, pp. 216–240. Springer, Cham (2017)
3. Abe, Y., Hayashi, Y.-i., Mizuki, T., Sone, H.: Five-Card AND protocol in committed format using only practical shuffles. In: APKC@AsiaCCS 2018. Ed. by K. Emura, J. H. Seo, and Y. Watanabe. ACM, pp. 3–8. (2018). https://doi.org/10.1145/3197507.3197510

4. Barrington, D.A.M.: Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. J. Comput. Syst. Sci. **38**(1), 150–164 (1989). https://doi.org/10.1016/0022-0000(89)90037-8
5. Biçer, O., Bingöl, M. A., Kiraz, M. S., Levi, A.: Towards Practical PFE: An Efficient 2-Party Private Function Evaluation Protocol Based on Half Gates. In: IACR Cryptology ePrint Archive. Cryptology ePrint Archive, Report 2017/415 (2017)
6. Bultel, X., Dreier, J., Dumas, J., Lafourcade, P., Miyahara, D., Mizuki, T., Nagao, A., Sasaki, T., Shinagawa, K., Sone, H.: Physical Zero-Knowledge Proof for Makaro. In: Stabilization, safety, and security of distributed systems, SSS 2018. Ed. by T. Izumi and P. Kuznetsov. LNCS. Springer, pp. 111–125. (2018). https://doi.org/10.1007/978-3-030-03232-68
7. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. P., Yang, K.: On the (Im)possibility of obfuscating programs. In: CRYPTO 2001. Ed. by J. Kilian. LNCS 2139. Springer, pp. 1–18 (2001). https://doi.org/10.1007/3-540-44647-81
8. den Boer, B.: More efficient match-making and satisfiability: the five card trick. In: EUROCRYPT '89. Ed. by J. Quisquater and J. Vandewalle. LNCS 434. Springer, pp. 208–217 (1989). https://doi.org/10.1007/3-540-46885-423
9. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS 2001. IEEE Computer Society, pp. 136–145 (2001). https://doi.org/10.1109/SFCS.2001.959888. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7601
10. Chaum, D., Carback, R., Clark, J., Essex, A., Popoveniuc, S., Rivest, R.L., Ryan, P.Y.A., Shen, E., Sherman, A.T., Vora, P.L.: Scantegrity II: end-to-end verifiability by voters of optical scan elections through confirmation codes. IEEE Trans. Inf. Forensics Secur. **4**(4), 611–627 (2009). https://doi.org/10.1109/TIFS.2009.2034919
11. Carback, R., Chaum, D., Clark, J., Conway, J., Essex, A., Herrnson, P. S., Mayberry, T., Popoveniuc, S., Rivest, R. L., Shen, E., Sherman, A. T., Vora, P. L.: Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy. In: USENIX Security Symposium 2010, Proceedings. USENIX Association, pp. 291–306 (2010). http://www.usenix.org/events/sec10/tech/full papers/Carback.pdf
12. Crépeau, C., Kilian, J.: Discreet solitary games. In: CRYPTO '93. Ed. by D. R. Stinson. LNCS 773. Springer, pp. 319–330 (1993). https://doi.org/10.1007/3-540-48329-227
13. Canetti, R., Vald, M.: Universally composable security with local adversaries. In: Visconti, I., De Prisco, R. (eds.) Security and Cryptography for Networks, SCN 2012. LNCS, vol. 7485. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-32928-9_16
14. Dvorák, P., Koucký, M.: Barrington Plays Cards: The Complexity of Card-based Protocols. In: LIPIcs 187 (2021). Ed. by M. Bläser and B. Monmege, 26:1–26:17. https://doi.org/10.4230/LIPIcs.STACS.2021.26
15. Dixon, J.D., Mortimer, B.: Permutation Groups. Graduate Texts in Mathematics; 163. Springer, New York (1996)
16. Durham, R.: Skipjack. In: Steven (ed.) Galbraith's Games (2015). https://www.thegamecrafter.com/games/skipjack
17. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS 2013. IEEE Computer Society, pp. 40–49 (2013). https://doi.org/10.1109/FOCS.2013.13
18. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: TCC 2010. Ed. by D. Micciancio. LNCS 5978. Springer, New York, pp. 308–326 (2010). https://doi.org/10.1007/978-3-642-11799-219
19. Günther, D., Kiss, Á., Schneider, T.: More efficient universal circuit constructions. In: ASIACRYPT 2017. Ed. by T. Takagi and T. Peyrin. LNCS 10625. Springer, pp. 443–470 (2017). https://doi.org/10.1007/978-3-319-70697-916
20. Goyal, R., Koppula, V., Waters, B.: Lockable obfuscation. In: FOCS 2017. Ed. by C. Umans. IEEE Computer Society, pp. 612–621 (2017). https://doi.org/10.1109/FOCS.2017.62
21. Gradwohl, R., Naor, M., Pinkas, B., Rothblum, G.: Cryptographic and physical zero-knowledge proof systems for solutions of sudoku puzzles. In: FUN 2007. Ed. by P. Crescenzi, G. Prencipe, and G. Pucci. LNCS 4475. Springer, pp. 166–182 (2007). https://doi.org/10.1007/978-3-540-72914-316. http://www.wisdom.weizmann.ac.il/~naor/PAPERS/sudokuabs.html
22. Hashimoto, Y., Shinagawa, K., Nuida, K., Inamura, M., Hanaoka, G.: Secure Grouping Protocol Using a Deck of Cards. In: ICITS 2017. Ed. by J. Shikata. LNCS 10681. Springer, New York, pp. 135–152 (2017). https://doi.org/10.1007/978-3-319-72089-08

23. Ishikawa, R., Chida, E., Mizuki, T.: Efficient card-based protocols for generating a hidden random permutation without fixed points. In: UCNC 2015. Ed. by C. S. Calude and M. J. Dinneen. LNCS 9252. Springer, pp. 215–226 (2015). https://doi.org/10.1007/978-3-319-21819-916
24. Katz, J.: Universally composable multi-party computation using tamper-proof hardware. In: EUROCRYPT 2007. Ed. by M. Naor. LNCS 4515. Springer, pp. 115–128 (2007). https://doi.org/10.1007/978-3-540-72540-47
25. Kastner, J., Koch, A., Walzer, S., Miyahara, D., Hayashi, Y.-i., Mizuki, T., Sone, H.: The Minimum Number of Cards in Practical Card-based Protocols. In: ASIACRYPT 2017. Ed. by T. Takagi and T. Peyrin. LNCS 10626. Springer, pp. 126–155 (2017). https://doi.org/10.1007/978-3-319-70700-65
26. Koch, A.: Cryptographic protocols from physical assumptions. PhD thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT) (2019). https://doi.org/10.5445/IR/1000097756
27. Koch, A.: The Landscape of Optimal Card-based Protocols. In: Journal of Mathematical Cryptology (Special Issue: Proceedings of MathCrypt 2021). In press
28. Koch, A.: The landscape of security from physical assumptions. In: IEEE Information Theory Workshop, ITW 2021. IEEE (2021). https://doi.org/10.1109/ITW48936.2021.9611501
29. Koch, A., Schrempp, M., Kirsten, M.: Card-based cryptography meets formal verification. In: ASIACRYPT 2019, Proceedings, Part I. Ed. by S. D. Galbraith and S. Moriai. LNCS. Springer, Nov. 25, pp. 488–517 (2019). https://doi.org/10.1007/978-3-030-34578-518
30. Koch, A., Walzer, S.: Foundations for actively secure cardbased cryptography. In: Fun with Algorithms, FUN 2021. Ed. by M. Farach-Colton, G. Prencipe, and R. Uehara. LIPIcs 157. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 17:1– 17:23 (2020). https://doi.org/10.4230/LIPIcs.FUN.2021.17
31. Koch, A., Walzer, S., Härtel, K.: Card-based Cryptographic Protocols Using a Minimal Number of Cards. In: ASIACRYPT 2015. Ed. by T. Iwata and J. H. Cheon. LNCS 9452. Springer, pp. 783–807 (2015). https://doi.org/10.1007/978-3-662-48797-632
32. Lipmaa, H., Mohassel, P., Sadeghian, S.: Valiant's universal circuit: improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017 (2016)
33. Mizuki, T., Asiedu, I. K., Sone, H.: Voting with a Logarithmic Number of Cards. In: UCNC 2013. Ed. by G. M. et al. LNCS 7956. Springer, pp. 162–173 (2013). https://doi.org/10.1007/978-3-642-39074-616
34. Miyahara, D., Hayashi, Y., Mizuki, T., Sone, H.: Practical and easy-to-understand card-based implementation of Yao's millionaire protocol. In: Combinatorial Optimization and Applications, COCOA 2018. Ed. by D. Kim, R. N. Uma, and A. Zelikovsky. LNCS. Springer, pp. 246–261 (2018). https://doi.org/10.1007/978-3-030-04651-417
35. Mizuki, T.: Efficient and secure multiparty computations using a standard deck of playing cards. In: CANS 2016. Ed. by S. Foresti and G. Persiano. LNCS 10052. pp. 484–499 (2016). https://doi.org/10.1007/978-3-319-48965-029
36. Moran, T., Naor, M.: Basing cryptographic protocols on tamper-evident seals. In: Theoretical Computer Science 411.10, pp. 1283–1310 (2010). https://doi.org/10.1016/j.tcs.2009.10.023
37. Mizuki, T., Sone, H.: Six-card secure AND and four-card secure XOR". In: FAW 2009. Ed. by X. Deng, J. E. Hopcroft, and J. Xue. LNCS 5598. Springer, pp. 358–369 (2009). https://doi.org/10.1007/978-3-642-02270-836
38. Mohassel, P., Sadeghian, S. S.: How to hide circuits in MPC an efficient framework for private function evaluation. In: EUROCRYPT 2013. Ed. by T. Johansson and P. Q. Nguyen. LNCS 7881. Springer, pp. 557–574 (2013). https://doi.org/10.1007/978-3-642-38348-933
39. Mizuki, T., Shizuya, H.: A formalization of card-based cryptographic protocols via abstract machine. In: International Journal of Information Security 13.1, pp. 15–23 (2014). https://doi.org/10.1007/s10207-013-0219-4
40. Nishida, T., Mizuki, T., Sone, H.: Securely computing the three-input majority function with eight cards. In: TPNC 2013. Ed. by A. H. Dediu, C. Martín-Vide, B. Truthe, and M. A. Vega-Rodríguez. LNCS 8273. Springer, pp. 193–204 (2013). https://doi.org/10.1007/978-3-642-45008-216
41. Niemi, V., Renvall, A.: Secure Multiparty Computations Without Computers. In: Theoretical Computer Science 191.1-2, pp. 173–183 (1998). https://doi.org/10.1016/S0304-3975(97)00107-2
42. Niemi, V., Renvall, A.: Solitaire Zero-knowledge. In: Fundam. Inform. 38.1-2, pp. 181–188 (1999). https://doi.org/10.3233/FI-1999-381214
43. Popoveniuc, S., Hosp, B.: An introduction to PunchScan. In: Towards Trustworthy Elections. Ed. by D. C. et al. LNCS 6000. Springer, pp. 242–259 (2010). https://doi.org/10.1007/978-3-642-12980-315

44. Sasaki, T., Mizuki, T., Sone, H.: Card-based zero-knowledge proof for sudoku. In: Fun with Algorithms, FUN 2018. Ed. by H. Ito, S. Leonardi, L. Pagli, and G. Prencipe. LIPIcs 100. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 29:1–29:10 (2018). https://doi.org/10.4230/LIPIcs.FUN.2018.29
45. Valiant, L. G.: Universal circuits (preliminary report). In: STOC 1976. Ed. by A. K. Chandra, D. Wotschke, E. P. Friedman, and M. A. Harrison. ACM, pp. 196–203 (1976). https://doi.org/10.1145/800113.803649
46. Verhoeff, T.: The zero-knowledge match maker. (2014). https://www.win.tue.nl/~wstomv/publications/liber-AMiCorum-arjeh-bijdrage-van-tom-verhoeff.pdf
47. Wichs, D., Zirdelis, G.: Obfuscating compute-and-compare programs under LWE. In: FOCS 2017. Ed. by C. Umans. IEEE Computer Society, pp. 600–611 (2017). https://doi.org/10.1109/FOCS.2017.61