A Practical Universal Circuit Construction and Secure Evaluation of Private Functions

Vladimir Kolesnikov[1] and Thomas Schneider[2*]

[1] Bell Laboratories, 600 Mountain Ave. Murray Hill, NJ 07974,USA
kolesnikov@research.bell-labs.com
[2] Dept. of Comp. Sci., University of Erlangen-Nuremberg, Germany
thomas.schneider@informatik.stud.uni-erlangen.de

Abstract. We consider general secure function evaluation (SFE) of *private functions* (PF-SFE). Recall, privacy of functions is often most efficiently achieved by general SFE [17,18,9] of a Universal Circuit (UC).
Our main contribution is a new simple and efficient UC construction. Our circuit UC_k, universal for circuits of k gates, has size $\sim 1.5k \log^2 k$ and depth $\sim k \log k$. It is up to 50% smaller than the best UC (of Valiant [15], of size $\sim 19k \log k$) for circuits of size up to ≈ 5000 gates.
Our improvement results in corresponding performance improvement of SFE of (small) private functions. Since, due to cost, only small circuits (i.e. < 5000 gates) are practical for PF-SFE, our construction appears to be the best fit for many practical PF-SFE.
We implement PF-SFE based on our UC and Fairplay SFE system [10].

Key words: SFE of private functions, universal circuit, privacy

1 Introduction

We consider two-party secure function evaluation (SFE) of *private functions* (PF-SFE). Recall, "regular" SFE techniques allow two parties to evaluate any function on their respective inputs x and y, while keeping the inputs secret. SFE is a subject of immense amount of research, e.g. [17,18,9]. Efficient SFE algorithms enable a variety of electronic transactions, previously impossible due to mutual mistrust of participants. Examples include auctions [11,3,5,1], contract signing [4], distributed database mining [7,8], etc. As computation and communication resources have increased, SFE became practical for common use. Fairplay [10] is a full implementation of generic two-party SFE with malicious players. It demonstrates feasibility and efficiency of SFE of practical functions, represented as circuits of up to $\approx 10^6$ gates. Today, generic SFE is a relatively mature technology, and even small improvements are non-trivial and welcome.

In this work, we impose an additional restriction on SFE. Namely, we require that the evaluated function is known only by one party and needs to be kept secret (i.e. everything besides the size, the number of inputs and the number of

[*] The work was done while the author was visiting Bell Laboratories.

outputs is hidden from the other party). Examples of real-life private functions include credit evaluation function, background- and medical history checking function, airport no-fly check function, etc. Full or even partial revelation of these functions opens vulnerabilities in the corresponding process, exploitable by dishonest participants (e.g. credit applicants), and should be prevented.

It is well known that the problem of PF-SFE can be reduced to the "regular" SFE [14,13]. This is done by parties evaluating a *Universal Circuit* (UC) instead of a circuit defining the evaluated function. UC can be thought of as a "program execution circuit", capable of simulating any circuit C of certain size, given the description of C as input. Therefore, disclosing the UC does not reveal anything about C, except its size. At the same time, the SFE computes output correctly and C remains private, since the player holding C simply treats description of C as additional (private) input to SFE. This reduction is the most common (and often the most efficient) way of securely evaluating private functions [14,13].

Our improvement of the UC construction directly results in improvements of PF-SFE for many practical private functions of interest. Indeed, circuit-based SFE (e.g. Yao's garbled circuit [17,18,9]) is still the most efficient SFE method for many important functions, such as the comparison function. The elegant and very efficient auction system of Naor, Pinkas and Sumner [11] implements auction function as a circuit, as well. Further, due to the size of UC constructions, PF-SFE is practical only for small circuits (UC for 5000-gate circuits has size 10^6, pushing the general SFE size limit). Therefore, improvements of circuit representation is particularly relevant for small circuits, and this is the focus and the result of our work.

1.1 Our contributions

Our main contribution is a new elegant and efficient universal circuit UC_k construction of size $\sim 1.5k \log^2 k$ and depth $\sim k \log k$. For the circuits most relevant for PF-SFE (of size up to ≈ 5000), our approach results in up to 50% size reduction compared to asymptotically optimal construction of Valiant [15]. See Table 1 in Sect. 5 for detailed comparison. As described above, this immediately implies improvement in the practical PF-SFE. We expand this discussion and present additional applications below in Sect. 1.3.

Our constructions are simple and practical. We used them to implement PF-SFE as an extension of the Fairplay SFE system [10].

The basic building blocks we developed (such as the efficient S_v^u selection blocks of Sect. 4.2) may be of use in other circuit constructions as well.

1.2 Related work

The most efficient known UC_k construction is the celebrated construction of Valiant [15]. With size $\sim 19k \log k$, it is asymptotically optimal, with a small constant factor. It relies on universal graphs. UC_k is derived from a universal graph UG_k; UC_k is universal for circuits of size k, if UG_k is universal for graphs of k nodes and in- and out-degrees 2. Embedding of the graph representation of a circuit C into UG_k defines the programming of UC_k to simulate C. As noted above, our construction produces smaller UC_k for circuits most relevant for PF-SFE. Further, we believe that implementation of our construction is more self-contained and straightforward.

Waksman [16] describes how to construct and program a permutation network, a circuit implementing an arbitrary permutation on n elements. Waksman's construction is asymptotically optimal (size $\sim 2n \log n$ and depth $\sim 2 \log n$). We use this work in an essential way – fundamental building blocks of our UC construction rely on [16].

1.3 Applications for universal circuits

As discussed above, UC is naturally used to extend the functionality or privacy in numerous practical SFE applications, in particular those based on Yao's garbled circuit [17,18,9]. Recall, Yao's approach views the evaluated function as a binary circuit known to both parties. The idea is to encrypt the signals on all wires of the circuit. Then the evaluator (one of the participants of the computation) uses clever setup and properties of encryption to compute (gate by gate) encryption of the output wires from the encryptions of input wires. The result of SFE is obtained by decrypting the values of the output wires of the circuit. We note that the cost of Yao's construction depends only on the size of the circuit, and not on its depth or fan-out. To perform PF-SFE, instead of evaluating the circuit directly, a UC that is programmed with the original circuit is evaluated. As UC can be programmed with any circuit, the evaluated function is entirely hidden from the evaluator.

We discuss natural applications that directly benefit from our improvements.

Frikken et. al [6] show a privacy-preserving credit checking scheme that is based on the evaluation of a garbled circuit. Their scheme is limited to the special class of credit-checking policies that can be expressed as the weighted sum of criteria. By evaluating a universal circuit their scheme can be extended to arbitrary, more complicated, private credit-checking policies.

Cachin et al. [2] describe autonomous mobile agents which migrate between several distrusting hosts. Garbled-circuit-based, their scheme ensures the privacy of the inputs of the visited hosts but not the structure of the mobile agent's code. The privacy of the executed code can be guaranteed by evaluating universal circuits instead.

Ostrovsky and Skeith [12] show how to filter remote streaming data (e.g airports' passenger lists, on-line news feeds or internet chat-rooms) using secret keywords and their combinations, such as no-fly lists. Their protocol allows Collector (e.g. airport) to obliviously filter out entries that match the (encrypted) query, which are then sent back for decryption. Their scheme can be naturally extended to allow a much finer private matching criteria, additionally preserving data privacy, as follows. The Collector encrypts each filtered stream element with a random pad. The querying party thus obtains the list of encrypted matches. In the second round, the querying party uses PF-SFE (e.g. using our UC_k) to search the matching data with an arbitrary, more detailed private search function.

2 Definitions and Preliminaries

In this section, we present basic notation and building blocks of our construction.

In the following, a *gate* is the implementation of a boolean function $\{0,1\}^2 \rightarrow \{0,1\}$ that has two *inputs* and one *output*. We consider acyclic *circuits* that consist of connected gates with arbitrary fanout, i.e. the (single) output of each gate can be used as input to an arbitrary number of gates. Further, each output of the circuit C is the output of a gate and not a redirected input of C.

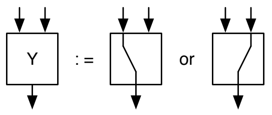
A *block* B_v^u is a circuit that has u inputs $in_1, .., in_u$ and v outputs $out_1, .., out_v$ (we always associate variable u with inputs and v with outputs). B_v^u computes a function $f_B : \{0,1\}^u \rightarrow \{0,1\}^v$ that maps the input values to the output values. For simplicity, we identify B_v^u with f_B and write: $B(in_1, \ldots, in_u) = (out_1, \ldots, out_v)$. The *size* of a block B, $size(B)$, is the number of gates B consists of; its depth, $depth(B)$, is the maximum number of gates between any input and any output of B. A block can be a sub-block of a larger block. We construct a circuit as a collection of functional blocks, as this simplifies presentation.

A *programmable* block is a block that consists of connected programmable gates with unspecified function tables. *Programming* a programmable block is done by providing a specific function table for each of its gates.

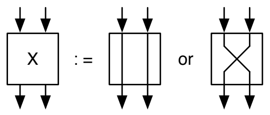
A *Universal Circuit* $\mathrm{UC}_{u,v,k}$ is a programmable block with u inputs and v outputs that can be programmed to simulate any circuit C with up to u inputs, v outputs and k gates. UC_C denotes UC that is programmed to simulate circuit C, that is $\forall(in_1, \ldots, in_u) : \mathrm{UC}_C(in_1, \ldots, in_u) = C(in_1, \ldots, in_u)$.

A *one-output switching block* Y is a programmable block that computes $(in_1, in_2) \rightarrow in_1$ or in_2, as shown in Fig. 1(a). It is implemented by one gate programmed with the corresponding function table. $size(Y) = depth(Y) = 1$.

A *two-output switching block* X is a programmable block shown on Fig. 1(b) that computes $(in_1, in_2) \rightarrow (in_1, in_2)$ or (in_2, in_1). It is implemented by using (in parallel) two Y blocks: one for each of the outputs. $size(X) = 2; depth(X) = 1$.



(a) Y switching block



(b) X switching block

Fig. 1. Switching blocks

A *selection block* S_v^u is a programmable block that selects for each of its v outputs one of the u input values (with duplicates). S_v^u is programmed according to the selection mapping $(\sigma_i)_{i=1}^v, \sigma_i \in \{1..u\}$ that selects the σ_i-th input as the i-th output. That is, a programmed S_v^u computes $S(in_1, \ldots, in_u) = (in_{\sigma_1}, \ldots, in_{\sigma_v})$.

A S_1^u selection block can be implemented by $(u-1)$ Y blocks that are programmed to switch the desired input value in_{σ_1} to the output. Shallow S_1^u is obtained by arranging Y blocks in a tree. Thus, $size(S_1^u) = u-1, depth(S_1^u) = \log u$.

A naive implementation of S_v^u selection block uses a S_1^u selection block for each of the v outputs, resulting in $size(S_v^u) = v(u-1)$ and $depth(S_v^u) = \log u$.

Selection blocks are crucial for our UC construction. We describe much more efficient S_v^u constructions in Sect. 4.2.

3 Our universal circuit construction

In this section, we present our modular UC construction. All of the necessary building blocks were introduced in Sect. 2; here we show how to assemble them. Then, in Sect. 4, we design improved versions of some building blocks, which results in performance improvement of our UC.

In our UC construction, we simulate each gate G_i of the original circuit C. That is, for each G_i, $\mathrm{UC}_{u,v,k}$ has a corresponding programmable G_i-simulation gate G_i^{Sim}. In our construction, we always ensure that inputs, outputs and semantics of G_i^{Sim} correspond to G_i. Additionally, we hide the wiring of C by ensuring that every possible wiring can be implemented in $\mathrm{UC}_{u,v,k}$. This is the natural method of construction of UC, and is, in fact, employed by Valiant [15].

We design our UC construction recursively (we build a circuit from two circuits of smaller size). We first note that the input/output interface of $\mathrm{UC}_{u,v,k}$ is different from that of the natural recursion step. This is why we introduce a *universal block* U_k. U_k can be viewed as a UC with specific input and output semantics. Namely, U_k has $2k$ inputs and k outputs, since this is a maximum $\mathrm{UC}_{u,v,k}$ can have. Further, we restrict that U_k's inputs in_{2i-1}, in_{2i} are only delivered to the simulation gate G_i^{Sim}, and U_k's i-th output comes from G_i^{Sim}. (Of course, input of some gates G_i may come from any other gates' outputs, and not from in_{2i-1} or in_{2i}, which may not be used at all. U_k allows this; it only restricts that G_i's input cannot come from other in_j). U_k is thus a UC for the class of circuits of size k with the above input/output restrictions.

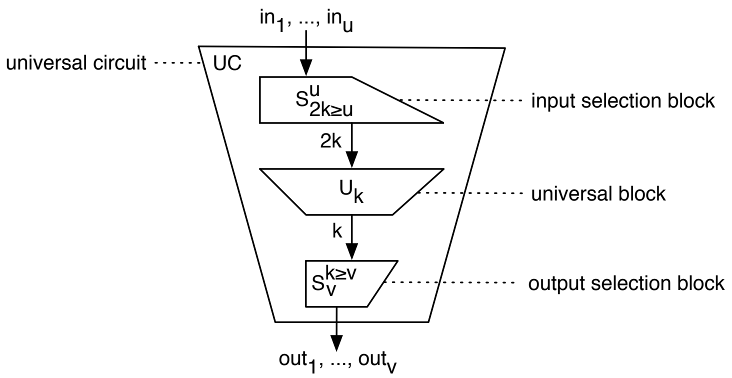


Fig. 2. Modular universal circuit construction

Now, given an implementation of U_k, it is easy to construct $\mathrm{UC}_{u,v,k}$ (shown on Fig. 2). We need to provide the input selection block, which directs inputs of UC to the proper inputs of U_k. Finally, we need the output selection block, directing outputs of U_k to the proper outputs of UC, and discarding unused outputs. Both blocks are instances of selection blocks discussed above.

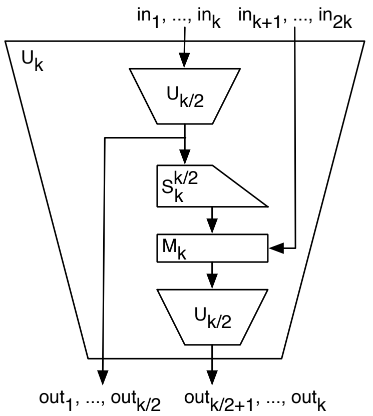
In the next section, we present our U_k construction. Plugged in the construction of Fig. 2, it gives a complete UC construction.

3.1 Recursive universal block construction

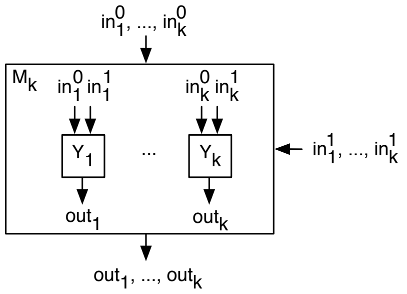
In this section, we describe the natural divide-and-conquer procedure for constructing U_k, capable of simulating any circuit C_k of size k, with the input/output restrictions mentioned above.

In the following, we refer to the gates of the circuit C_k by their index. We choose a topological order of the gates $G_1, \ldots, G_k$, which ensures that the i-th gate G_i has no inputs that are outputs of a successive gate G_j, where $j > i$. Since we only consider acyclic circuits, we can always obtain this ordering by topological sorting with complexity $O(k)$.

Now, suppose we have two blocks $\mathrm{U}_{k/2}$, universal for circuits $C_{k/2}$ of size $k/2$. We would like to combine them to obtain U_k. Clearly, because of their universality, one of $\mathrm{U}_{k/2}$ could simulate the "upper" half of C_k (i.e. gates G_1 through $G_{k/2}$) , and the other $\mathrm{U}_{k/2}$ could simulate the lower half (gates $G_{k/2+1}, \ldots, G_k$). Note, by the topological ordering, there is no data going into the upper $\mathrm{U}_{k/2}$ from the lower one. Thus, U_k must only direct its inputs/outputs and allow implementation of all possible data paths from the upper $\mathrm{U}_{k/2}$ to the lower one. This can be naturally done, as shown on Fig. 3(a). We describe this in detail below.



(a) Recursive construction of U_k



(b) Mixing block M_k

Fig. 3. Recursive universal block construction

The first k inputs to U_k $in_1, .., in_k$ are directly sent to the upper $\mathrm{U}_{k/2}$. Note, the order of the inputs matches the interface perfectly, so no additional manipulation is required. The $k/2$ outputs of the upper (resp. lower) $\mathrm{U}_{k/2}$ are sent directly to the first (resp. second) half of the outputs of U_k. Again, interfaces match, and no manipulation is required.

We now only need to show how the inputs to the lower $\mathrm{U}_{k/2}$ are provided. These inputs could come from (any G_i^{Sim} gate of) the upper $\mathrm{U}_{k/2}$. Therefore, we also wire the outputs of upper U$_{k/2}$ into a selection block $S_k^{k/2}$. This allows to direct, with duplicates, the output of any gate of upper U$_{k/2}$ to any position of the input interface of lower U$_{k/2}$ (and thus to any gate of lower U$_{k/2}$). Additionally, (some of) lower U$_{k/2}$'s inputs could come from the U$_k$ inputs $in_{k+1}, ...in_{2k}$. Since the lower U$_{k/2}$ simulates gates $G_{k/2+1}$ through G_k of C_k, inputs $in_{k+1}, ...in_{2k}$ are already ordered to match lower U$_{k/2}$'s interface. Now, for each input of lower U$_{k/2}$, we need to switch between the two input wires: one provided by upper U$_{k/2}$ via $S_k^{k/2}$, and the other coming from U$_k$'s input directly. This is easily achieved by a Y switching block. On the diagram, for ease of presentation, we combine the k of these Y blocks into a *mixing block* M_k, shown on Fig. 3(b) with $size(M_k) = k \cdot size(Y) = k$ and $depth(M_k) = 1$.

The base case of the recursive construction is U_1, a universal block implementing a single gate. U_1 is implemented by a single programmable gate. This completes the description of the recursive U$_k$ construction.

The above immediately implies efficient methods of UC programming, given the circuit C_k. In particular, if the first (resp. second) input of a gate G_j in the lower half of C_k $(k/2 < j \leq k)$ is connected to an input of C_k, the mixing block M_k is programmed to select the corresponding input in_{2j-1} (resp. in_{2j}) of U$_k$ by programming Y_{2j-k-1} (resp. Y_{2j-k}) of M_k correspondingly (see Fig. 3(b)). Otherwise, if G_j is connected to an output of a gate G_i in the upper half of C_k $(1 \leq i \leq k/2)$, M_k and $S_k^{k/2}$ are programmed to select the corresponding output from the upper $U_{k/2}$ block by programming Y_{2j-k-1} (resp. Y_{2j-k}) correspondingly and programming $S_k^{k/2}$ with $\sigma_{2j-k-1} = i$ (resp. $\sigma_{2j-k} = i$).

We now compute the complexity of our constructions U$_k$ and UC (using selection block constructions of Sect. 4.2). Recall, the cost of Yao's garbled circuit depends only on its size, and not on depth. Note, $size(U_1) = 1, depth(U_1) = 1$.

$$\begin{aligned}
size(U_k) &= 2size(U_{k/2}) + size(S_k^{k/2}) + size(M_k)\\
&= k \cdot size(U_1) + \sum_{i=0}^{\log(k)-1} 2^i(size(S_{k/2^i}^{k/2^{i+1}}) + size(M_{k/2^i}))\\
&= k + \sum_{i=0}^{\log(k)-1} 2^i(6\frac{k}{2^{i+1}}\log(\frac{k}{2^{i+1}}) + 3 + \frac{k}{2^i})\\
&= k + 3k\log^2 k - 2k\log k - 3k\sum_{i=0}^{\log(k)-1} i + 3\sum_{i=0}^{\log(k)-1} 2^i\\
&= k + 3k\log^2 k - 2k\log k - 3k \cdot 0.5(\log(k)(\log(k) - 1)) + 3(k - 1)\\
&= 1.5k\log^2 k - 0.5k\log k + 4k - 3;\\
depth(U_k) &= 2depth(U_{k/2}) + depth(S_k^{k/2}) + depth(M_{2k}) = \dots\\
&= k\log k + k + 4\log k - 12.
\end{aligned}$$

Using the optimization of Sect. 4.3, U_k has complexity $size(U_k) = 1.5k\log^2 k - 1.5k\log k + 6k - 5$ and $depth(U_k) = k\log k + 4\log k - 11$.

U_k combined with input- and output-selection blocks of Sect. 4.2 as shown in Fig. 2, results in a UC construction of complexity

$$\begin{aligned} size(UC) &= 1.5k\log^2 k + 2.5k\log k + 9k + (u+2k)\log u + (k+3v)\log v \\ &\quad -2u - 4v + 1; \\ depth(UC) &= k\log k + 2k + v + 7\log k + 2\log u + 3\log v - 14. \end{aligned}$$

4 Improved selection block constructions

In this section, we present efficient selection block S_v^u constructions. They can be plugged directly in our UC construction. The size and depth computation of UC presented in Sect. 3.1, uses efficient constructions of this section.

We start the presentation with two useful generalizations of the permutation blocks of Waksman [16]. Based on these, we construct efficient selection blocks which are directly used in our UC construction.

4.1 Generalized permutation blocks

$\mathbf{P_u^u}$ permutation block. A *permutation block* P_u^u is a programmable block that can be programmed to output any permutation of the inputs. Formally, given a permutation $(\pi_i)_{i=1}^u, \pi_i \in \{1, \ldots, u\}, \forall i \neq j : \pi_i \neq \pi_j$ that selects for the i-th output a unique input π_i, P_u^u computes $P(in_1, .., in_u) = (in_{\pi_1}, .., in_{\pi_u})$.

When u is a power of 2, Waksman [16] describes an efficient recursive P_u^u construction built from X switching blocks. His P_u^u has $size(P_u^u) = 2u\log u - 2u + 2$ and $depth(P_u^u) = 2\log u - 1$.

Waksman also gives an efficient recursive algorithm to program the X switching blocks of his construction. (Fig. 4 describes a slight generalization of Waksman's construction; fixing $u = v$ in Fig. 4 corresponds to Waksman's P_u^u.) The programming algorithm takes a $u \times u$ permutation matrix for the permutation (π_i) as input. It splits this $u \times u$ permutation matrix into two $u/2 \times u/2$ permutation matrices that are recursively implemented by the left and the right $P_{u/2}^{u/2}$ permutation sub-block and programs the X switching blocks correspondingly. Using a sparse matrix representation for the permutation matrices this algorithm can be efficiently implemented in $O(u\log u)$.

We note that Waksman's construction can be naturally generalized to the cases where $u \neq v$, i.e. the number of inputs and outputs differ. Below we define the resulting objects (which we call "truncated permutation" and "expanded permutation" blocks), and present their efficient constructions.

$\mathbf{TP_v^{u \geq v}}$ truncated permutation block. A $TP_v^{u\geq v}$ truncated permutation block permutes a subset of v of the u inputs to the $v \leq u$ outputs. The remaining $u - v$ input values are discarded. Formally, an output mapping $(\mu_i)_{i=1}^v$, $\mu_i \in \{1, \ldots, u\}, \forall j \neq i : \mu_i \neq \mu_j$ selects the μ_i-th input as the i-ths output. The truncated permutation block computes $TP(in_1, \ldots, in_u) = (in_{\mu_1}, \ldots, in_{\mu_v})$.

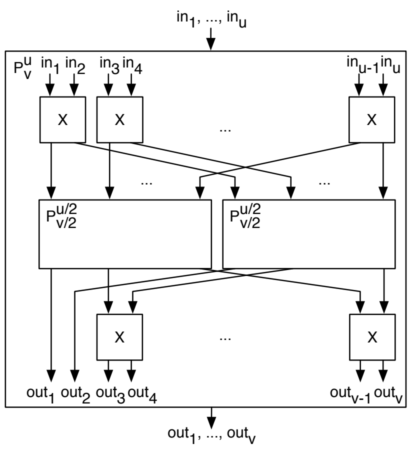


Fig. 4. Recursive construction of a P_v^u permutation block

The $TP_v^{u \geq v}$ block is recursively constructed analogous to Waksman's permutation network construction as seen in Fig. 4. W.l.o.g we assume u and v are even at each recursion step (otherwise we introduce an unused dummy input or output with small overhead). If $u \geq 2$ the $TP_v^{u \geq v}$ truncated permutation block is divided into two $TP_{v/2}^{u/2 \geq v/2}$ truncated permutation sub-blocks. The upper $u/2$ X switching blocks distribute the inputs of $TP_v^{u \geq v}$ to the two sub-blocks. The lower $(v/2 - 1)$ X switching blocks distribute the outputs of the two sub-blocks to the outputs of $TP_v^{u \geq v}$ as shown in Fig. 4. At the base of the recursion, if $v = 1$, a S_1^u selection block selects the intended input.

The $TP_v^{u \geq v}$ block is programmed using a natural generalization of Waksman's recursive programming algorithm. The intended output mapping (μ_i) is expressed as a $u \times v$ truncated permutation matrix. In each recursion step the algorithm splits the $u \times v$ matrix into two $u/2 \times v/2$ truncated permutation matrices implemented by the left and right sub-block and programs the X switching blocks accordingly. In the end of the recursion, if the truncated permutation matrix is a $u \times 1$ matrix with a one in the i-th row, the S_1^u selection block is programmed to select the i-th input value as output: $\sigma_1 = i$. This algorithm can be implemented in $O((u + v) \log v)$ using sparse matrix representations.

The complexity of this construction is $size(TP_v^{u \geq v}) = (u+v) \log v + u - 3v + 2$ and $depth(TP_v^{u \geq v}) = \log u + \log v - 1$.

$\mathbf{EP_{v \geq u}^u}$ expanded permutation block. An $EP_{v \geq u}^u$ expanded permutation block permutes the u inputs to a subset of u of the $v \geq u$ outputs. The remaining $v - u$ outputs are allowed to obtain any input value (they are intended to be later discarded and are called *dummy* outputs). Formally, an input mapping $(\mu_i)_{i=1}^u, \mu_i \in \{1, \ldots, v\}, \forall j \neq i : \mu_i \neq \mu_j$ specifies that the i-th input should be mapped to the μ_i-th distinct output. The expanded permutation block computes $EP(in_1, \ldots, in_u) = (out_1, \ldots, out_v)$ where $(out_s = in_r) \leftrightarrow (\mu_r = s), s \in \{1, \ldots, v\}, r \in \{1, \ldots, u\}$.

The construction of the $EP^u_{v\geq u}$ is analogous to the previously described $TP^{u\geq v}_v$ block. At the base of the recursion, if $u = 1$, the single input in_1 is connected to each of the v outputs.

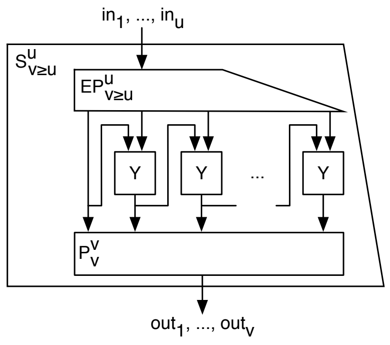
The programming algorithm of $EP^u_{v\geq u}$ is analogous to that of $TP^{u\geq v}_v$ as well. The input is a $u \times v$ matrix that corresponds to (μ_i) and it can be implemented in $O((u+v)\log u)$.

The construction has complexity $size(EP^u_{v\geq u}) = (u+v)\log u - 2u + 2$ and $depth(EP^u_{v\geq u}) = 2\log u$.

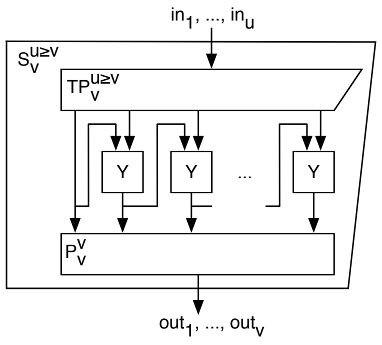
4.2 Efficient selection blocks

We use truncated and expanded permutation blocks of the previous section to build efficient selection blocks $S^u_{v\geq u}$, used directly in the UC construction.

$\mathbf{S^u_{v\geq u}}$ selection block. We obtain the $S^u_{v\geq u}$ selection block from one $EP^u_{v\geq u}$ expanded permutation block, one P^v_v permutation block, and $(v-1)$ Y switching blocks as shown in Fig. 5(a).



(a) $S^u_{v\geq u}$ selection block



(b) $S^{u\geq v}_v$ selection block

Fig. 5. S^u_v selection blocks

It is not hard to see that the above $S^u_{v\geq u}$ is indeed a selection block, i.e. it can be programmed with any selection mapping $(\sigma_i)^v_{i=1}, \sigma_i \in \{1,\ldots,u\}$. To program $S^u_{v\geq u}$, first count the frequency of occurrence c_j of each input value in the output: $c_j = \#\{\sigma_i : \sigma_i = j; i \in \{1\ldots v\}\}; j \in \{1\ldots u\}$. Note, $0 \leq c_j \leq v$ and $\sum_{j=1}^u c_j = v$. The $EP^u_{v\geq u}$ expanded permutation block is programmed to

1. map the needed inputs $(c_j \neq 0)$ to its $(\sum_{k=1}^{j-1} c_k)$-th output and
2. map the unused inputs $(c_j = 0)$ to an unused (dummy) output.

The $(v-1)$ Y switching blocks connected to the outputs of $EP^u_{v\geq u}$ duplicate the needed inputs as necessary and feed them to the P^v_v permutation block. They are programmed as follows. If the right input of a Y block is a needed output (produced by Step 1), then the Y block selects it as output. Otherwise, the output of the neighbor Y block is selected. For each j, this construction inputs c_j copies of in_j into the P^v_v permutation block. P^v_v then permutes these values to the corresponding outputs indicated by the selection mapping (σ_i). The complexity of this construction is $size(S^u_{v\geq u}) = (u+v)\log u + 2v\log v - 2u - v + 3$ and $depth(S^u_{v\geq u}) = 2\log u + 2\log v + v - 2$.

Permutation-based $\mathbf{S^{u\geq v}_v}$ selection block. An efficient $S^{u\geq v}_v$ selection block can be constructed and programmed analogously, but using a $TP^{u\geq v}_v$ truncated permutation block instead as shown in Fig. 5(b). Its complexity is $size(S^{u\geq v}_v) = (u+3v)\log v + u - 4v + 3$ and $depth(S^{u\geq v}_v) = \log u + 3\log v + v - 3$.

Improved $\mathbf{S^u_{2u}}$ selection block. In this section, we optimize the $S^u_{v\geq u}$ selection block construction for the case $v = 2u$, most frequently used in our recursive construction of the universal block U_k. We improve by replacing the $EP^u_{v\geq u}$ expanded permutation block in the construction of $S^u_{v\geq u}$ in Fig. 5(a) with a smaller P^u_u permutation block and a different connection of the $(v-1)$ Y blocks as shown in Fig. 6. Our construction achieves $size(S^u_{2u}) = 6u\log u + 3$ and $depth(S^u_{2u}) = 4\log u + 2u - 1$.

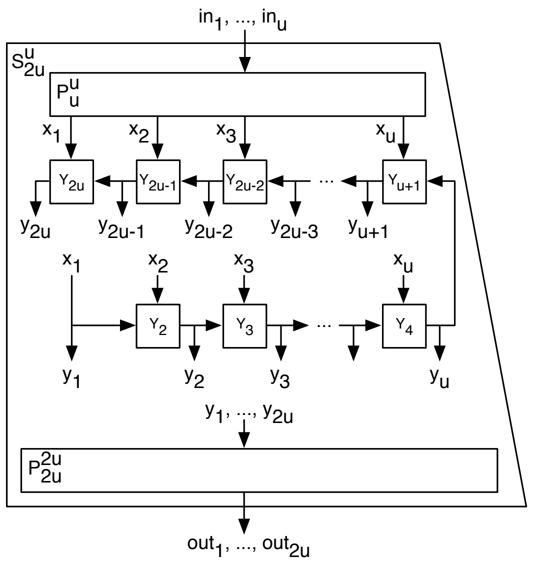


Fig. 6. Improved S^u_{2u} selection block

Lemma 1. *Construction of Fig. 6 is a S^u_{2u} selection block.*

Proof. To prove Lemma 1, we only need to show that the upper permutation block P^u_u together with the layer of Y blocks output the selected values (with the right number of duplicates each) in some order. (The rest, i.e. imposing the desired order, is done by the lower permutation block P^{2u}_{2u}.)

We use the network of Y blocks to duplicate (or omit) inputs as required by the selection block specification. The upper permutation block P^u_u can be programmed to deliver the desired input in_i to any Y-layer input x_j not already used by another input. For example, if input in_i needs to be duplicated c_i times, this can be achieved by programming the permutation to map in_i to x_j, and have blocks Y_j through Y_{j+c_i-1} to output x_j. This way, as required, the value in_i would be duplicated c_i times.

For efficiency reasons, the wiring of the Y-layer is limited. In particular, input x_i is delivered only to blocks Y_i and Y_{2u-i+1}, which are in column i. From there, x_i can be propagated "to the right" from Y_i (i.e. to blocks $Y_{i+1}, ...,$ in the lower row) and/or "to the left" from Y_{2u-i+1} (i.e. to blocks $Y_{2u-i+2}, ...,$ in the upper row). Note, blocks Y_i and Y_{2u-i+1} cannot receive different inputs from P_u^u. They, however, can produce different outputs, since one or both of them could be propagating the value of their neighbouring Y block.

It is not immediately clear that the inputs $in_1 ... in_u$ can be permuted such that the Y-layer can provide the right number of duplicates for each input. We show, that this in fact can be done. We observe that this permutation and the Y-layer programming can be reduced to the following box-packing problem.

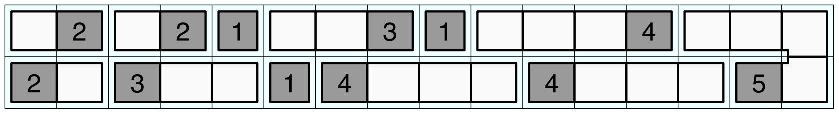


Fig. 7. Valid arrangement of boxes produced by Algorithm 1 for boxes of size $(c_j) = \{2, 3, 1, 4, 4, 5, 4, 1, 3, 1, 2, 2, 0, 0, 0, 0\}$. Dark gray head cells contain size.

Box-packing. (See Fig. 7 for illustration.) There are u rectangular boxes of sizes $c_1, \ldots, c_u$, where $c_i \in \{0, \ldots, 2u\}$ and $\sum_{i=1}^{u} c_i = 2u$. Each non-empty i-th box consists of a head cell (dark gray), and $c_i - 1$ trailing cells (light gray). There is a rectangular $2 \times u$ grid of slots that consists of an upper row and a lower row. A box of size c_i occupies c_i consecutive slots in one row (one exception is that the right-most box might wrap around from the lower to the upper row, as seen on Fig. 7). The boxes in the upper row are oriented with heads to the right, and the boxes in the lower row are oriented with heads to the left. A *collision* occurs when two heads occupy slots in the same column. The arrangement of all u boxes is called *valid*, if it contains no collisions. (Note that a valid arrangement leaves no empty slots.) A solution to the box-packing problem is a valid arrangement.

A procedure for a valid arrangement of the boxes of sizes $c_1, \ldots, c_u$ gives the following natural programming of the P_u^u permutation block and the Y-layer. Associate (1-to-1) each input in_i of size c_i with a box of same size c_i and compute a valid arrangement. Then, input in_i is switched by P_u^u to x_j if the j-th column is occupied by the head of the box associated with in_i. Inputs in_i with $c_i = 0$ (unused inputs) are switched to the columns j which have no head boxes. Both switching blocks Y_i and Y_{2u-i+1} of each column i are programmed as follows. They select input x_i iff the corresponding slot in the valid arrangement is occupied by the head (otherwise, the output of the neighbored Y switching block is selected). It is not hard to see that this programming results in the desired output, given the corresponding valid arrangement of boxes.

Lemma 2 below shows an efficient box-packing procedure. This completes the proof of Lemma 1. □

Algorithm 1 *(Box-packing)*

```
0. Each box is always put in the leftmost unoccupied slots in the specified row.
1. Sort boxes by size in increasing order.
2. while there is at least one box of size 1, do
   (a) if there are at least two boxes of minimal sizes $s_2 \geq s_1 \geq 2$ left
         i. put the box of size $s_1$ in the upper row
        ii. put remaining (but no more than $s_1$-2) boxes of size 1 in lower row
       iii. put the box of size $s_2$ in the lower row (possibly wrap around)
        iv. put remaining (but no more than $s_2$-2) boxes of size 1 in upper row
   (b) else // there is only one box of size $s_1 \geq 2$ left
         i. put the remaining boxes of size 1 in the lower row
        ii. put the box of size $s_1 \geq 2$ in the lower row and wrap around
3. while there is at least one box of minimal size $s_3 \geq 2$ left, do
   (a) if there is another box of minimal size $s_4 \geq s_3 \geq 2$ left
         i. put the box of size $s_3$ in the upper row
        ii. put the box of size $s_4$ in the lower row (possibly wrap around)
   (b) else // there is only one box of size $s_3 \geq 2$ left
         i. put the box of size $s_3 \geq 2$ in the lower row and wrap around
4. end
```

Lemma 2. *Algorithm 1 efficiently produces a valid arrangement for any given set of u boxes of sizes $c_1, \ldots, c_u; 0 \leq c_j \leq 2u; \sum_{j=1}^{u} c_j = 2u$.*

Proof. Note, since $\sum c_j = 2u$, for each box of size $2 + i$, there must be i boxes of size 1, or $i/2$ boxes of size 0, or a corresponding combination.

A) Algorithm 1 always puts all boxes and *terminates*. We first show that Step 2 eliminates all boxes of size 1. Indeed, suppose the contrary, a block of size 1 remains. Then, in each previous execution of Step 2a, we eliminated blocks of sizes $s_2 \geq s_1 \geq 2$ and $s_1 + s_2 - 4$ blocks of size 1, and in Step 2b we eliminated a block of size s_1 and $s_1 - 2$ blocks of size 1. Since $\sum c_j = 2u$, there could not have been more blocks of size 1 than we eliminated, and we arrive at contradiction. Further, Step 3 eliminates all remaining boxes of size ≥ 2. In each iteration, at least one box of size $s_3 \geq 2$ is eliminated either in Step 3(a)i or Step 3(b)i, until all boxes of size ≥ 2 are eliminated. (Observe, at each iteration, upper row "grows" not more than the lower. Thus, Algorithm's actions are always legal.)

B) Algorithm 1 produces a *valid* arrangement. We need to show that no step of Algorithm 1 causes a collision. It is easy to see that Step 2a and Step 2b never cause a collision. Further, once Step 2 has finished, the number of occupied slots in the upper row ω_{up} is less or equal to the number of occupied slots in the lower row ω_{down}, with $0 \leq \omega_{down} - \omega_{up} \leq s_2 - 2$ (here s_2 is the size of the most recently put block in Step 2(a)iv). Since the boxes are processed in increasing order, in Step 3, $s_3 \geq s_2 \geq 2$. If the box of size s_3 is the last remaining one, it is put in the lower row in Step 3(b)i and, as is easy to see, doesn't cause a collision. Otherwise, in Step 3(a)i, the box of size s_3 is put in the upper row. The number of occupied slots in the upper row is now $\omega'_{up} = \omega_{up} + s_3$, and the upper row has at least two more occupied slots than the lower row: $\omega'_{up} - \omega_{down} = (\omega_{up} + s_3) - \omega_{down} \geq 2$.

This implies that the next Step 3(a)ii doesn't cause a collision when putting the box of length $s_4 \geq s_3$ into the lower row. After Step 3(a)ii, the number of occupied slots in the lower row is $\omega'_{down} = \omega_{down} + s_4$. In the end of the current iteration of Step 3, the number of occupied slots in the upper row is again less or equal to the number of occupied slots in the lower row: $\omega'_{down} - \omega'_{up} = (\omega_{down} + s_4) - (\omega_{up} + s_3) = (\omega_{down} - \omega_{up}) + (s_4 - s_3) \geq 0$ and hence the length relationship between the upper and lower rows ($0 \leq \omega'_{down} - \omega'_{up} \leq s_4 - 2$) is the invariant of Step 3. Therefore, no iteration of Step 3 causes a collision. As no step causes a collision, Algorithm 1 produces a valid arrangement.

C) Algorithm 1 is *efficient*. Sorting of the u boxes in Step 1 costs $O(u \log u)$. Steps 2 and 3 have a runtime of $O(u)$, as in every iteration at least one box is eliminated. Hence the runtime of Algorithm 1 is in $O(u \log u)$. □

4.3 Optimization of the universal circuit construction

As the order of the two inputs of a gate simulation block G can be swapped by swapping its function table, we can omit the last row of X blocks in the lower P_k^k permutation block of the $S_k^{k/2}$ selection block in the construction of U_k (see Fig. 3(a), Fig. 6 and Fig. 4) and adapt the programming correspondingly. This results in a reduction of $\Delta size(U_k) = k \log k - 2k + 2$ and $\Delta depth(U_k) = k - 1$.

5 Comparison and conclusion

We now compare our UC solution to the best previously known Valiant's UC [15]. Recall, we consider circuits $\mathrm{UC}_{u,v,k}$, universal for circuits of k gates, u inputs and v outputs. Valiant's UC has $size(\mathrm{UC}^{Valiant}_{u,v,k}) = (19k + 9.5u + 9.5v) \log k + O(k)$ and ours has $size(\mathrm{UC}_{u,v,k}) = 1.5k \log^2 k + 2.5k \log k + (u + 2k) \log u + (k + 3v) \log v + O(k)$. To help visualize the relationship, Table 1 shows sample relative sizes of our UC construction compared to Valiant's: $size_{rel} = \frac{size(\mathrm{UC}_{u,v,k})}{size(\mathrm{UC}^{Valiant}_{u,v,k})}$. We also note the break-even point $k_{eq} = k|_{size_{rel}=1}$ — the maximum size of circuits for which our UC is smaller.

Table 1. Comparison between our and Valiant's UC construction [15].

circuit inputs and outputs			break-even	relative size $size_{rel}$		
	u	v	point k_{eq}	$k = 1,000$	$k = 5,000$	$k = 10,000$
few	$o(k)$	$o(k)$	$2,048$	91.8%	110.2%	118.1%
	$0.5k$	$0.1k$	$5,000$	86.0%	100.1%	106.2%
	$0.5k$	$0.25k$	$8,000$	83.1%	96.4%	102.1%
	$1k$	$0.5k$	$117,000$	69.0%	79.5%	84.0%
many	$2k$	$1k$	$26,663,000$	53.6%	60.9%	64.1%

While Valiant's construction is asymptotically better, our UC is up to 50% smaller for small circuits, due to much lower constant factors. For PF-SFE, small circuits are of most interest, since only they can be evaluated efficiently today (indeed, UC for 5000-gate circuits has size $\approx 10^6$). In addition, our construction is more detailed and seems to be much easier to implement than Valiant's. Thus, we believe that our UC construction is a good fit for *practical* PF-SFE. In support of our contribution, we have successfully implemented general PF-SFE based on the Fairplay system [10] and our UC construction.

Acknowledgements. We thank reviewers of FC'08 for helpful comments.

References

1. Ian F. Blake and Vladimir Kolesnikov. Conditional encrypted mapping and comparing encrypted numbers. In *Financial Cryptography and Data Security, FC 2006*, volume 4107 of *LNCS*, pages 206–220. Springer, 2006.
2. Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In *ICALP '00*, pages 512–523, London, UK, 2000. Springer-Verlag.
3. Giovanni Di Crescenzo. Private selective payment protocols. In *Financial Cryptography and Data Security, FC 2000*, volume 1962 of *LNCS*. Springer, 2000.
4. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.
5. Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In *RSA Security 2001 Cryptographer's Track*, volume 2020 of *LNCS*, pages 457–471. Springer-Verlag, 2001.
6. Keith Frikken, Mikhail Atallah, and Chen Zhang. Privacy-preserving credit checking. In *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, pages 147–154, New York, NY, USA, 2005. ACM Press.
7. Murat Kantarcioglu and Chris Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. In *ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'02)*, 2002.
8. Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *LNCS*, pages 20–24. Springer, 2000.
9. Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004.
10. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX*, 2004.
11. Moni Naor, Benny Pinkas, and Reuben Sumner. Privacy preserving auctions and mechanism design. In *1st ACM Conf. on Electronic Commerce*, 1999.
12. Rafail Ostrovsky and William E. Skeith III. Private searching on streaming data. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *LNCS*, 2005.
13. Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, 4(2):12–19, 2002.
14. Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC^1. In *Proc. 40th IEEE Symp. on Foundations of Comp. Science*, pages 554–566, New York, 1999. IEEE.
15. Leslie G. Valiant. Universal circuits (preliminary report). In *Proc. 8th ACM Symp. on Theory of Computing*, pages 196–203, New York, NY, USA, 1976. ACM Press.
16. Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.
17. Andrew C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 160–164, Chicago, 1982. IEEE.
18. Andrew C. Yao. How to generate and exchange secrets. In *Proc. 27th IEEE Symp. on Foundations of Comp. Science*, pages 162–167, Toronto, 1986. IEEE.